SYMANTEC™

THINK C™

User's Manual

Fastest Development Time
Compile in seconds, link instantly

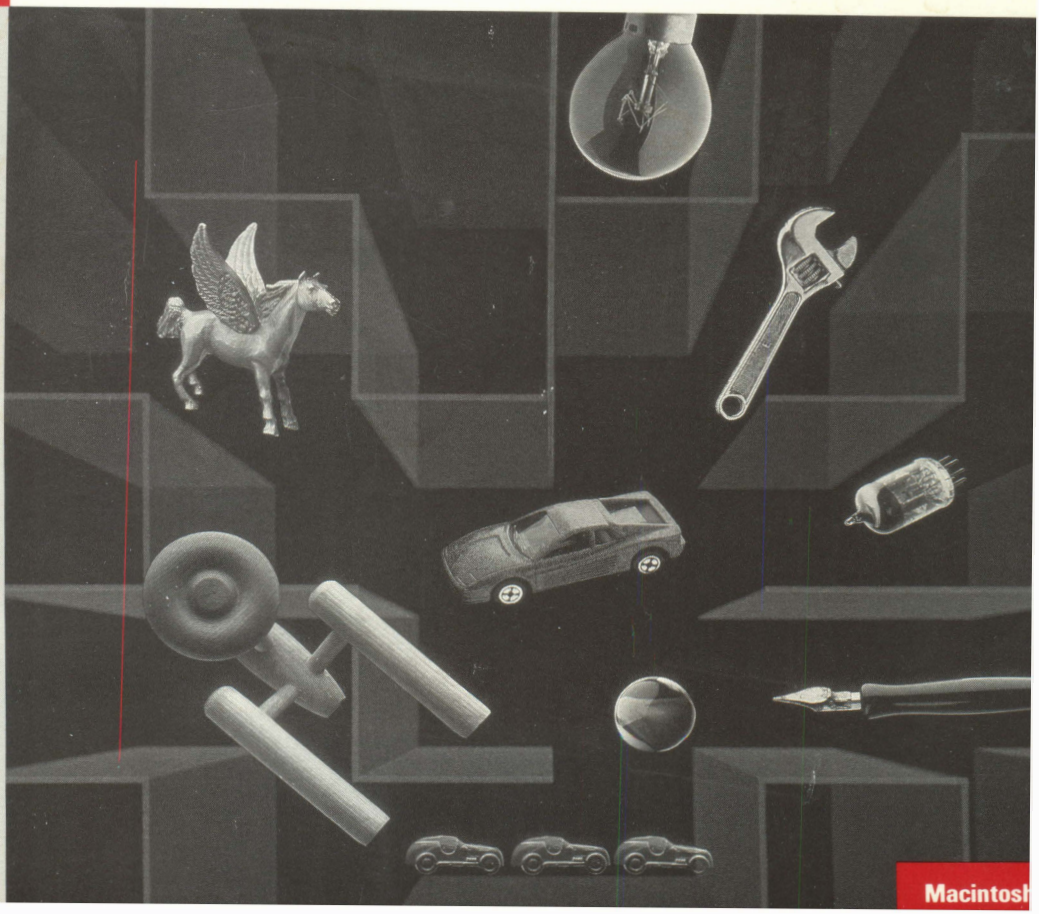
Full Source-Level Debugging
Set break points, trace execution, examine and modify all variables including objects

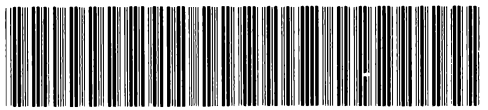
Integrated Development Environment
Edit, compile, link, debug and run programs in one environment

Professional-Quality Code Generation
Fast compiler creates extremely compact, commercial- quality code

Object-Oriented Programming Support
Object extensions for flexible, extensible and reusable code

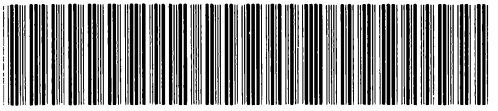
Powerful Class Library
Standard Macintosh user interface building blocks for quickly developing object-oriented applications





05000-26300-12734-71207

IMPORTANT - Save this card as proof of purchase for future product upgrades. When an upgrade is available, you will be notified through an upgrade mailing on how to use this card.
You must send in the attached registration card to be eligible.



05000-26300-12734-71207

IMPORTANT - DO NOT REMOVE THIS CARD
Your registration number is unique to your Symantec software package. We suggest that you keep this card inside your manual, so that it's always handy should you need to call us.

Change of Name and/or Address Card

Please use this postcard to notify Symantec if you change your address or name so that we may continue to keep you informed of any product updates and maintain our customer files.

Please check the appropriate box below:

☐ Address Change

☐ Name Change

Effective — —

First Name

Middle Initial

Last Name

Previous Last Name (if Name Change)

Company

Street

City State

Zip - Country

Daytime Phone Number

Ext.

05000-26300-12734-71207

SYMANTEC.

Disk Replacement Form

After your 90-Day Limited Warranty, if your disk(s) becomes unusable, you may fill out and return 1) this card, 2) your damaged disk(s) and your check or money order (see pricing on other side, add sales tax if applicable), to Symantec and receive a replacement disk(s). DURING THE 90-DAY LIMITED WARRANTY PERIOD, THIS SERVICE IS FREE. You must be a registered customer in order to receive disk replacements. If you have not yet signed and returned your Registration Card, you may return it with this card.

Product Name Version

Disk Format Operating System

Date Purchased - -

Last Name

First Name

Title

Company

Address

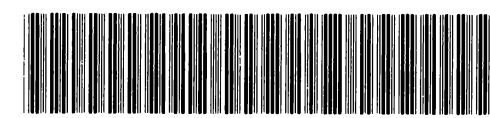
City State

Zip - Country

Daytime Phone

Ext

Briefly describe the problem_____



05000-26300-12734-71207

SYMANTEC.

IMPORTANT - READ REVERSE SIDE

Disk Replacement Price	$10.00
Sales Tax (if applicable, see below)	
Shipping and Handling	$5.00
TOTAL	

FORM OF PAYMENT (CHECK ONE)

☐ Check (Payable to Symantec) Amount enclosed $_____

☐ Visa #_____ Exp._____

☐ Mastercard #_____ Exp._____

☐ American Express #_____ Exp._____

Name on Card (please print)_____

Signature_____

SALES TAX INFORMATION We are required by law to collect sales taxes on shipments to the following states where we are located. Please add the appropriate amount. **CALIFORNIA** - 6% (plus applicable local tax). **GEORGIA** - 3% (plus applicable local tax). **IOWA** - 4% (plus applicable local tax). **MARYLAND** - 5%. **MASSACHUSETTS** - 5% . **MICHIGAN** - 4%. **MISSOURI** - 4.225% (plus applicable local tax). **NEW JERSEY**- 6% . **NEW YORK** - 4% (plus applicable local tax. **TEXAS** - 6% (plus applicable local tax). **WASHINGTON** - 6.5% (plus applicable local tax).

Please allow at least 3-4 weeks for delivery.

Send to: Symantec Corporation
Attn: Disk Replacement
10201 Torre Avenue
Cupertino, CA 95014

SYMANTEC.

IMPORTANT - READ REVERSE SIDE

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 706 CUPERTINO CA

POSTAGE WILL BE PAID BY ADDRESSEE

SYMANTEC™
Attn: Customer Registration
10201 Torre Avenue
Cupertino, CA 95014-9854

SYMANTEC™

THINK C™

User's Manual

Fastest Development Time
Compile in seconds, link instantly

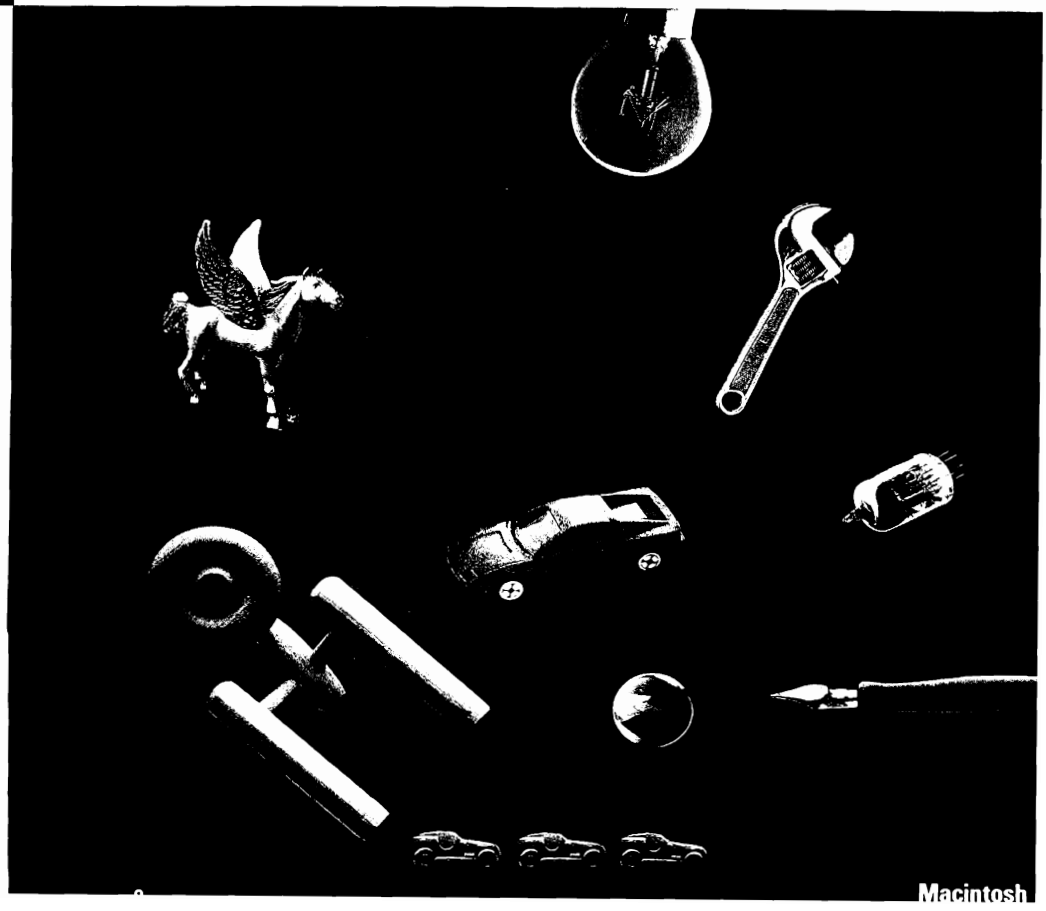
Full Source-Level Debugging
Set break points, trace execution, examine and modify all variables including objects

Integrated Development Environment
Edit, compile, link, debug and run programs in one environment

Professional-Quality Code Generation
Fast multi-pass compiler creates extremely compact, commercial-quality code

Object-Oriented Programming Support
Object extensions for flexible, extensible and reusable code

Powerful Class Library
Standard Macintosh user interface building blocks for quickly developing object-oriented applications



Credits

User's Manual: Philip Borenstein and Jeff Mattson

Software: Michael Kahl and Jörg 'jbx' Brown

Quality Assurance: Paul Vetri, Chris Morin, Greg Howe

Product Manager: Diana Bury

THINK Class Library: Gregory H. Dow

Copyright © 1989 Symantec Corporation. All Rights Reserved.

Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014
408/253-9600

Technical Support: For technical support in North America, call (617) 275-1710. In the United Kingdom telephone Symantec (UK) Ltd. directly at (0628) 776343. Outside North America or the UK please contact the exclusive Symantec distributor in your local area.

The product names mentioned in this manual are the trademarks or registered trademarks of their manufacturers.

ResEdit and RMaker are copyrighted programs of Apple Computer, Inc. licensed to Symantec Corp. to distribute for use only in combination with THINK C. Apple software shall not be copied onto another diskette (except for archive purposes) or into memory unless as part of execution of THINK C. When THINK C has completed execution, Apple Software shall not be used by any other program.

APPLE COMPUTER, INC. MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED REGARDING THE ENCLOSED SOFTWARE PACKAGE, IT'S MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED IN SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE THAT VARY FROM STATE TO STATE.

Contents

ONE GETTING STARTED

1 Welcome 1
What is THINK C? 1
What You Need 2
What's in the Package 3
What's in the Manual 3
What You Should Know 5
Notes for Experienced Users 8

2 Installing THINK C 13
Before You Do Anything Else... 13
Installing THINK C on a Hard Disk System 13
Installing on a Floppy System 14
Disk Layout Diagram 16

TWO LEARNING THINK C

3 Tutorial: Hello World 19
Creating the Project 19
Creating the Source File 22
Compiling the Source File 23
Adding the Library 25
Running the Project 26
Creating the Application 27
Where to Go Next 28

4 Tutorial: MiniEdit 29
Creating the Project 30
Adding the Source Files and Libraries 32
Compiling and Running the Project 34
Fixing a Bug 35
Running the Project Again 37
Building the Application 38
Using a Resource File 40
Finishing Up 40
Where to Go Next 40

5 Tutorial: Bullseye....41
Opening the Bullseye Project....42
Turning the Debugger On....42
Watching the Program Run....43
Examining and setting variables....49
Examining structs and arrays....54
Expressions and contexts....57
Quitting the Debugger....59

THREE USING THINK C

6 Overview....63
The THINK C Environment....63
The Project....63
Writing a Program in THINK C....64
Using THINK C....65

7 The Project....67
Anatomy of a Project....68
Segmentation....72
Building Applications....74
Building Desk Accessories and Device Drivers....76
Building Code Resources....84

8 The Editor....93
Creating and Opening Files....93
Editing a File....96
Printing Files....98
Closing and Saving Files....98
Searching and Replacing....99
Searching for a Pattern (Grep)....103

9 Files & Folders....109
How THINK C Names Files....110
How THINK C Looks for #include Files....110
Moving Files Within a Project....111
Using the Trees....112
Disk Layout Diagram....114

10 The Compiler....115
Compiling Source Files....115
Precompiled Headers....117
Calling the Macintosh Toolbox Routines....119
Code Generation Options....127
Compiler Options....129
Function Prototypes....130
Portability....131

11 The Debugger 135
The Debugger Windows 137
Working with the Source Window 139
Setting Breakpoints 142
Controlling Execution 144
Working with the Data Window 147
Using Low Level Debuggers 151
Quitting the Debugger 152
Memory Considerations 152

12 Assembly Language 155
Using the Inline Assembler 155
Differences from Other Assemblers 163
C Calling Conventions 166
Pascal Calling Conventions 168
Tips 170

13 Libraries 171
Using libraries 171
Creating Libraries 171
Converting object files into libraries 172

FOUR OBJECT-ORIENTED PROGRAMMING IN THINK C

14 Object-Oriented Programming 177
Objects and Messages 177
Classes 178
Inheritance and Polymorphism 179
Objects and the Macintosh Interface 180
Working with Objects 181
Where to Go Next 182

15 Using Objects in THINK C 183
Overview 183
Declaring a Class 184
Declaring and Using Objects 186
Defining and Using Methods 188
Direct Classes 189
Tips and Techniques 190
Summary 194

16 The THINK Class Library 195
Overview 196
Installing the THINK Class Library 201
Writing an Application with the THINK Class Library 203
Working with Panes 207
Working with Menus 215
Handling Low Memory Situations 219
Undoing and Mouse Tracking 219
THINK Class Library Resources 221
Modifying the THINK Class Library 223
Where to Go Next 223

17 CApplication 225
18 CBartender 241
19 CBorder 253
20 CBureaucrat 255
21 CButton 259
22 CCheckBox 261
23 CChore 263
24 CClipboard 265
25 CCluster 271
26 CCollection 275
27 CControl 277
28 CDataFile 283
29 CDecorator 287
30 CDesktop 289
31 CDirector 295
32 CDocument 299
33 CEditText 307
34 CEnvironment 311
35 CError 313
36 CFile 317
37 CList 321
38 CMouseTask 325
39 CObject 329
40 CPane 331

41 CPanorama....343
42 CPicture....349
43 CPrinter....353
44 CRadioButton....357
45 CRadioGroup....359
46 CScrollBar....361
47 CScrollPane....365
48 CSizeBox....369
49 CStaticText....371
50 CSwitchboard....377
51 CTask....381
52 CView....385
53 CWindow....393
54 Global Variables....399

FIVE REFERENCE

55 THINK C Menus....405
The Menu....405
The File Menu....406
The Edit Menu....409
The Search Menu....415
The Project Menu....418
The Source Menu....424
Windows Menu....427

56 Debugger Menus....429
The Menu....429
The File Menu....429
The Edit Menu....430
The Debug Menu....431
The Source Menu....433
The Data Menu....434
The Windows Menu....436

57 Language Reference....437

SIX APPENDICES

A The Profiler 445
Using the Profiler 445
Modifying the Profiler 446
Summary 446

B Troubleshooting 449
Getting Help 449
Some Common Problems 449

C Error Messages 453

D RMaker Reference 483
Using RMaker 483
RMaker File Format 483
Predefined Resource Types 485

Index 491

License Agreement 511

THINK C

PART ONE

Getting Started

1 Welcome
2 Installing THINK C

Welcome
1

Introduction

Welcome to THINK C. This chapter tells you what's in your THINK C package, what equipment you need, and what you need to know to write C programs on your Macintosh.

If you don't read manuals

If you need to get started quickly, read this chapter, the next chapter, and one of the tutorials.

If you're an experienced THINK C user

If you already use THINK C, you'll be pleased with the object extensions, the THINK Class Library and other improvements. Read "Notes for Experienced Users" at the end of this chapter.

Topics covered in this chapter:

- What is THINK C?
- What you need
- What's in the package
- What's in the manual
- What you should know
- Notes for experienced users

What is THINK C?

THINK C is a unique development environment for the Macintosh. It features a very fast compiler, a faster linker, an integrated text editor, an auto-make facility, and a project organizer that holds all the pieces together. Because the editor, the compiler, and the linker are all components of the same application, THINK C knows when edited source files need to be recompiled. And if you edit an #include file, the auto-make facility recompiles all the source files that depend on it for declarations.

With THINK C you can build Macintosh applications, desk accessories, device drives, and any kind of code resource. The standard C libraries include all the functions specified in the ANSI C standard, as well as some additional Unix operating system functions.

You can run your program from THINK C as you work on it. Your program runs exactly as if you had opened it from the Finder, not under a simulated environment. And if you use MultiFinder your program runs in its own partition while THINK C remains active, so you can examine and edit your source files as you watch your program run.

The THINK C development environment includes a source level debugger that lets you debug your code exactly how you wrote it. No more translating assembly language back into C. The debugger lets you set breakpoints, step through your code, debug objects, examine variables, and change their values while your program is running. And because the debugger runs under MultiFinder, you can edit your source files while you're debugging.

What You Need

THINK C works best when you have at least 2 megabytes (Mb) of RAM and a hard disk. It will also run with 1Mb of RAM and two 800K floppy drives. With only 1Mb you won't be able to take advantage of MultiFinder, and you won't be able to use the debugger. With a floppy based system, you won't be able to write very large programs.

How much RAM?

You can run THINK C on a Macintosh Plus, the Macintosh SE series, or the Macintosh II series. You can run THINK C on a Macintosh 512Ke if you've upgraded it to at least 1Mb of RAM.

You can run THINK C without the debugger, on any Macintosh computer with at least 1Mb of RAM. To use the debugger, you need at least 2Mb of RAM.

How much disk space?

The complete THINK C system takes up about 1.5Mb on your disk, not including your own files. The actual size of your system may be smaller, depending on the kinds of programs you work on.

Although you can use THINK C with two 800K floppy drives, it works much better when you use a hard disk.

Which System/Finder?

Use the latest System and Finder provided by Apple. At press time, this is System Tools 6.0.2 or 6.0.3 (System 6.0.2 or 6.0.3/Finder 6.1). THINK C requires at least System Tools 5.0 (System 4.2/Finder 6.0).

THINK C is designed to work best under MultiFinder. If you're using a Macintosh with 1Mb RAM, however, you're better off *not* using MultiFinder.

What's in the Package

Your THINK C package consists of four double sided floppies, this manual, and the *Standard Libraries Reference.*

What's in the Manual

This manual is organized in six sections: Getting Started, Learning THINK C, Using THINK C, Reference, and the Appendices. Each chapter begins with an introduction that describes what's in the chapter followed by a list of the major topics covered in the chapter.

Getting Started

This is the section you're reading. It contains this chapter and the installation instructions. Even if you don't read manuals, be sure to read the installation instructions in Chapter 2.

Learning THINK C

This section contains three tutorials. The first one, "Hello World" shows you how to write a minimal program that uses the standard C libraries and introduces you to the basics of using THINK C.

The second tutorial, "MiniEdit," shows you how to build a Macintosh application. This tutorial is based on the Sample program in *Inside Macintosh.* It shows you how to fix bugs, how to use resource files, and how to build a double-clickable application.

The third tutorial, "Bullseye," shows you how to use THINK C's source level debugger. Read this chapter even if you already know how to use THINK C.

Using THINK C

This section contains eight chapters that describe the different components of THINK C.

Overview, Chapter 6, describes how THINK C works.

The Project, Chapter 7, describes the four different kinds of projects. It gives you the details of building applications, desk accessories, device drivers, and code resources. This chapter contains several code examples to make writing your program easier.

The Editor, Chapter 8, describes the THINK C integrated text editor. The editor has several features to make editing C source files easier and a sophisticated searching facility.

Files and Folders, Chapter 9, tells you why you should follow the installation instructions in Chapter 2. This chapter describes how THINK C looks for files on your disk and how it names files.

The Compiler, Chapter 10, describes how THINK C compiles your source files. It also tells you how to call the Macintosh Toolbox routines, how to generate code for the 68881 floating point coprocessor, how to use function prototypes, and how to port code from Unix machines.

The Debugger, Chapter 11, describes the source level debugger. This chapter tells you how to control execution, how to set breakpoints, and how to examine and modify your variables as you debug.

Assembly Language, Chapter 12, describes THINK C's inline assembler. This chapter also explains C and Pascal calling conventions so your assembly language routines will integrate smoothly with both your C functions and the Macintosh Toolbox routines.

Libraries, Chapter 13, tells you how to build and use libraries in your THINK C programs, and how to convert object code from other compilers and assemblers into THINK C libraries.

Object-Oriented Programming in THINK C

This section contains three chapters that explain how to use object-oriented programming techniques with THINK C.

Object-Oriented Programming, Chapter 14, is a general introduction to object-oriented programming.

Using Objects in THINK C, Chapter 15, describes how to use THINK C's object-oriented programming extensions in your projects.

The THINK Class Library, Chapter 16, describes how to use the THINK Class Library.

Chapters 17–54, describe each class of the THINK Class Library.

Reference

This section contains three reference chapters.

THINK C Menus, Chapter 55, describes the THINK C menu commands.

Debugger Menus, Chapter 56, describes the source level debugger's menu commands.

Language Reference, Chapter 57, is a supplement to the C Language Reference (Appendix A) of Kernighan and Ritchie's *The C Programming Language, Second Edition*.

Appendices This section contains appendices that describe the code profiler, some tips and troubleshooting suggestions, a list of error messages (with explanations), the RMaker resource compiler reference, and the index.

Conventions in the Manual

The names of menus and commands are in **bold face**. When a technical term or key word is introduced, it also appears in bold face.

Names of files, code fragments, resource names, function names, and variables appear in "`typewriter face.`"

All numbers are decimal. Hexadecimal numbers are written in C notation: `0x3EFA` instead of Pascal notation (`$3EFA`).

In this manual, the term **Toolbox routine** means any routine in ROM. The Macintosh ROM actually consists two different kinds of routines: Operating System routines and Toolbox routines. Operating System routines deal with low-level aspects of the machine like the file manager, the event posting mechanism, interrupts, device management, etc. The Toolbox deals with high-level aspects like the drawing environment, the window mechanism, menus, dialogs, etc.

What You Should Know

This manual assumes you already know, or are at least learning, how to program in C. If you're just getting started in C, THINK C is a great platform.

If you're planning to write Macintosh applications, you should be familiar with the Macintosh Toolbox as described in *Inside Macintosh*. The Toolbox is the set of operating system and user interface routines that make a Macintosh a Macintosh. It's beyond the scope of this manual to show you how the different parts of the Toolbox work together.

Learning C

As the popularity of C grows, more and more introductory level books appear on the shelves. Some books assume that you're just learning how to program, and others assume that you already know how to program in another language. Some books spend time telling

you how to use the development environment: the editor, the linker, the make facility. These things are done very differently in THINK C, so when you choose a book, choose one that doesn't dwell to much on these aspects of programming.

If you're learning C from a book, or if you're using THINK C to do coursework, be sure to do the first tutorial. It shows you how to set things up to write and run C programs that use the standard C libraries.

The standard references for the C programming language are Kernighan & Ritchie's *The C Programming Language, Second Edition* (Prentice Hall) and Harbison & Steele's *C: A Reference Manual* (Prentice Hall). The *The C Programming Language, Second Edition* is an update to *The C Programming Language* that incorporates what was then the latest draft of the ANSI standard. These books assume that you're already an experienced programmer.

Software Engineering in C (Springer-Verlag) by Peter Darnell and Philip Margolis is an excellent introduction to the C programming language. This book is ideal for new C programmer's who have programmed in other languages.

C Traps and Pitfalls (Addison-Wesley) by Andrew Koenig is a good book for intermediate and advanced C programmers. It contains a detailed discussion of common C programing problems.

Standard C (Microsoft Press) by P. J. Plauger and Jim Brodie is a guide to writing C programs that conform to the ANSI C standard. Both of the authors were officers of the committee that drafted the ANSI standard.

Numerical Recipes in C (Cambridge University Press) by Press, Flannery, Teukolsky and Vetterling is a detailed technical description of numerical methods with implementation examples in C.

Learning to write Macintosh programs

If you're new to programming the Macintosh, you might find yourself overwhelmed by the complexity of the Macintosh Toolbox and unfamiliar programming techniques. When the Macintosh was released in 1984, there was very little technical information available to casual programmers, and even commercial developers had a hard time figuring out how to get things to work correctly.

The Macintosh is even more complex today than it was in 1984, but now there are more places you can go for information. Several good books introduce programming the Macintosh and teach some of the finer points of using the Macintosh Toolbox. No matter which books you choose to get started, *Inside Macintosh* is indispensable.

Inside Macintosh Volumes I-V (Addison-Wesley) is the official reference that describes the more than 900 Macintosh Toolbox routines. You might be able to get by without it for a while, but if you're planning to write serious applications, you just can't do without. At five

volumes, it represents a hefty investment. The first three volumes cover the fundamentals. Volumes IV and V cover the additions and changes made with the introduction of the Mac Plus, Macintosh SE, and Macintosh II.

In addition to *Inside Macintosh*, Apple also publishes these books through Addison-Wesley. *Programmers Introduction to the Macintosh Family* is a brief introduction to the fundamental aspects of Macintosh development and discusses the structure of the typical Macintosh application. *Technical Introduction to the Macintosh Family* is a more hardware-oriented discussion of Macintosh computers. The *Apple Numerics Manual* contains detailed information about the Standard Apple Numerics Engine (SANE). *Human Interface Guidelines: The Apple Desktop Interface* talks about the general design of a Macintosh interface.

The *Macintosh Programming Primer: Inside the Toolbox Using THINK's LightspeedC* (Addison-Wesley) by Dave Mark and Cartwright Reed is a good introduction to Macintosh programming for those already familiar with C. This tutorial explains how to use the Toolbox, handle resources, and write a Macintosh application. Also, the examples use some of the newer parts of the Macintosh system, such as the Notification Manager and HyperCard XCMDs and XFCNs. Best of all, the examples are all written for THINK C. They were written for version 3.0 but will still run under version 4.0.

Stephen Chernicoff's two volume set, *Macintosh Revealed* (Hayden Books), is another step-by-step introduction to Macintosh programming. Chernicoff shows you how to build a working application and points out the parts of *Inside Macintosh* you really need to know as opposed to the parts you just need to be aware of. The programs in the book are written in Pascal, but they're not too difficult to translate to THINK C.

Scott Knaster is the author of two books on Macintosh programming. The first, *How to Write Macintosh Software* (Hayden Books), teaches you what goes on inside the Toolbox. This book contains some valuable tips about debugging Macintosh programs. The second book, *Macintosh Programming Secrets* (Addison-Wesley), deals with some of the conventions and techniques that have become standard in Macintosh programs. It also contains information about the Macintosh II and the Mac SE. These books are more technical than *Macintosh Revealed* and are loaded with pictures, diagrams, and examples.

Finally, MacTutor is the leading technical journal for Macintosh programming. The articles range from tutorial examples to advanced techniques. MacTutor covers several languages, not just C, but most of the C examples are written in THINK C. (All of the programs described in the magazine are available on disk.)

Apple Programmer's and Developer's Association

The Apple Programmer's and Developer's Association (APDA) is Apple's in-house membership organization that distributes technical information to programmers and developers. APDA is a great source for Technical Notes, programming utilities, reference books, and information about announced (but unreleased) products. Membership costs $20 per year.

For information about membership and products, contact APDA directly:

Apple Programmer's and Developers Association (APDA)
Apple Computer, Inc.
20525 Mariani Avenue, Mail Stop 33G
Cupertino, CA 95014-6299

(800) 282-2732 (USA) (800) 637-0029 (Canada) (408) 562-3910 (Other)

TechAlliance

TechAlliance is an independent support group of Apple product users. Formerly, it was A.P.P.L.E. Co-op. TechAlliance publishes the *MacTech Quarterly* four times a year, runs a co-op buying program through which members receive rebates on their purchases, and offers technical support through a Referral Network, a hot line, and a computer bulletin board. A one-year membership to TechAlliance, including a year's subscription to *MacTech Quarterly*, costs $35. A year's subscription to *MacTech Quarterly* without a membership costs $21.

For more information, contact TechAlliance directly at:

TechAlliance
290 SW 43rd Street
Renton, WA 98055

(800) 245-8999 (USA, except Washington state) (800) 257-7562 (Washington state)
(800) 635-6642 (Canada) (206) 251-5222 (Other)

CompuServe

Symantec has a forum on CompuServe specifically for THINK C users. Simply type `GO SYMANTEC` at any ! prompt. You'll find discussions here about programming in general and THINK C in particular. The data libraries contain utilities as well as sources for many programs. When upgrades or patches to THINK C are ready, they're posted here first.

CompuServe also has an Apple developers forum. Just type `GO APPDEV` at any ! prompt. This forum is a good place to get in touch with the Macintosh programming community.

Notes for Experienced Users

If you've used THINK C before, you'll be pleased with the new features of this release. This section describes the changes and enhancements.

Upgrading THINK C

To upgrade your copy of THINK C, backup and delete all the files that came in the THINK C package and upgrades (but not your own projects and source files). If you're upgrading from THINK C 3.0 or later, that would include at least these files:

- THINK C (the application)
- THINK C Debugger
- relConv
- oConv
- MacHeaders
- Mac #includes folder
- Mac Libraries folder
- Libraries folder
- DA Shell

Then follow the instructions in the next chapter, "Installing THINK C." Since this upgrade includes many new or revised libraries, you should reinstall everything from scratch.

> **Note:** Be sure to copy your serial number from your registration card on the inside front cover of this manual (or some other place where you can be sure your won't lose it). You'll need it to get future upgrades or technical support.

Compatibility with earlier releases

THINK C 4.0 automatically reads and converts projects created with THINK C 2.0 and later. All object code will be removed, so you will need to recompile the project. THINK C 4.0 can use libraries created with THINK C 2.0 and later. But be sure to use the new version of `MacTraps` and all other supplied libraries.

You will not be able to use projects or libraries from versions earlier than 2.0. Instead, create a new project and add your existing source files.

Object extensions to C

The set of object extensions to C is the most significant addition to THINK C. These extensions support object-oriented programming. The extensions are comparable to Object Pascal as defined by Apple (available with THINK Pascal), but the syntax resembles C++. The extensions are upward-compatible with C++ (i.e., C++ compilers can compile THINK C programs, but THINK C cannot compile all C++ programs). Read Chapter 15 for more information.

The THINK Class Library

The THINK Class Library is a collection of classes that implement the core of a standard Macintosh application. The THINK Class Library implements all the standard features of a

Macintosh application and makes writing Macintosh applications easier. Chapter 16 talks about the THINK Class Library, and Chapters 17–54 describe each class in detail.

More ANSI C compatibility

Many new language features in this release make THINK C more compatible with the ANSI C standard. Although THINK C is close to the standard, it is not conformant as defined in the standard. The remaining issues either are of little significance or would require fundamental changes to THINK C. For more information on ANSI compatibility, read the Chapter 57.

Some of the new language features for ANSI compatibility are:

- **The ANSI library.** THINK C's new `ANSI` library implements the full set of functions required by ANSI. Applications using the library create a simple console window that emulates a glass terminal. You'll find it useful for porting from UNIX or MS-DOS. The `ANSI` library replaces many of the UNIX libraries used in previous versions of THINK C. For an example of an application that uses the `ANSI` library, see Chapter 3. For more detailed information, see the *Standard Libraries Reference.*

- **New-style function definitions.** You may use function prototype syntax for function definitions as well as declarations.

- **The defined() preprocessor operator.** You may use the `defined()` preprocessor operator within `#if` and `#elif` directives.

- **# and ## preprocessor operators.** The # operator followed by a macro argument expands into the argument's value enclosed in double quotes. The ## operator concatenates its arguments together.

- **The long double type.** You can use the long double type. It is the same size as double. Floating-point literals suffixed with an `L` are long double literals.

Inline assembler extensions

The inline assembler now accepts all MC68020 and MC68881 instructions and addressing modes. For more information see Chapter 12, "Assembly Language."

Other changes

Many other features are new. Here are some of the most significant:

- **Once-only headers.** You can define a header file that's #included in many files but processed only once. For more information, see "Once-only headers" in Chapter 9.

- **Auto-make checks libraries.** The auto-make facility will reload a library if it was changed since it was last loaded. For more information, see Chapter 13, "Libraries."

- **Inline function definitions.** You can define inline functions. See the sections "Defining inline functions" and "Defining MC68881 unary inline functions" in Chapter 10.

- **Improved code resources.** Code resources can be multi-segment. Also, when building a code resource, you can merge it into an existing file. Read the section "Building Code Resources" in Chapter 7 for more information.

- **Support for debugging control panel devices.** You can use the source-level debugger to debug your control panel device (sometimes called a cdev). See the `Read Me` file in the folder `cdev stuff` for more information.

Installing THINK C
2

Introduction

This chapter tells you how to install THINK C on your Macintosh.

Topics covered in this chapter:

- Registering your copy of THINK C
- Installing on a hard disk system
- Installing on a floppy system
- Recommended disk layout

Before You Do Anything Else...

Write the serial number for your copy of THINK C onto the inside front cover of this manual (or some other place where you can be sure you won't lose it). You'll find the serial number on your registration card. Also, fill out and send the registration card so we can contact you about upgrades and other news.

> **Note:** Be sure to write down that serial number! You'll need it to get future upgrades or technical support.

Installing THINK C on a Hard Disk System

This section tells you how to set up THINK C on your hard disk. This setup ensures that THINK C will know where to find all the files it needs to compile your programs. To learn more about why the files are organized this way, see Chapter 9.

Installation summary

First, you'll create a development folder. The development folder will contain a folder for THINK C, the #include files, and the libraries. Then you'll create folders for each of your projects. The project folders will be in the development folder, but not in the THINK C folder.

The picture at the end of this chapter shows you what this disk layout looks like. You can use the picture to set up your disk, or you follow these directions.

Installation instructions

Start at the Finder and create a new folder. Name it `Development`.

Open the Development folder and create a new folder in it. Call the new folder THINK C Folder.

Now, copy all the files from disk THINK C 1 to the THINK C Folder. At least these files should be in the THINK C Folder: (Note that for THINK C to work correctly, these files must be in the same folder.)

- THINK C (the application)
- THINK C Debugger
- MacHeaders
- Mac #includes folder
- Mac Libraries folder
- oops Libraries

Next, copy the C Libraries folder from THINK C 2 to the THINK C Folder. This folder contains the standard C libraries, the standard #include files, and the library sources.

If you will be using the THINK Class Library, copy the THINK Class Library folder from THINK C 3 to the THINK C Folder. Then, copy the TCL Demos folder from THINK C 2 to the Development folder (not the THINK C Folder). The TCL Demos folder contains a project, Starter.π, that you'll use as a framework for projects you build with the THNK Class Library. Chapter 16 gives you more details about the files you need to use the THINK Class Library.

> **Note:** Be sure that you copy the TCL Demos folder to the Development folder and not the THINK C Folder.

That's all there is to it. Be sure to read Chapter 9 to learn more about how THINK C treats files and folders.

Installing on a Floppy System

Although THINK C works best when you use a hard disk, it's possible to use it if you have two 800K floppy drives. If you use floppies, you probably like to put your application and a System folder on one floppy and all of your data files on another. Unfortunately, *the THINK* C system won't fit on the same floppy as the System. To use THINK C from floppies, your "data disk" will be your System disk, and THINK C will be on another floppy.

> **Note:** Because of its large size, you shouldn't use the THINK Class Library on a floppy system.

Making the System disk

First, create your System disk. Just drag the System folder from one of your original Macintosh System disks to a blank floppy.

Next, use the Font/DA Mover to remove all the fonts except the ones needed by the system. Just select all the font names, and click on the Remove button. You'll get a warning dialog telling you the system fonts won't be removed. While you're still in the Font/DA Mover, remove all but one of the Desk Accessories. (The Calculator seems to be the smallest one).

Now remove the print drivers, the Control Panel files (General, Mouse, Sound, etc.). Leave the System, Finder, MultiFinder, and DA Handler.

You'll use this System floppy as your data disk. Use this disk to store your projects.

Making the THINK C disk

Now you're ready to make the THINK C disk. This disk will contain the THINK C application, the libraries, and the #include files. This will be a minimal working set, containing only those files you absolutely need for your project. You may need to create different THINK C disks for each project.

First, copy these files from `THINK C 1` to a blank floppy:

- `THINK C`
- `MacHeaders`
- `MacTraps` from the `Mac Libraries` folder.
- `oops Libraries`

If you are planning on writing programs that use the ANSI library, copy these files from the `C Libraries` folder on the `THINK C 2` disk to your floppy:

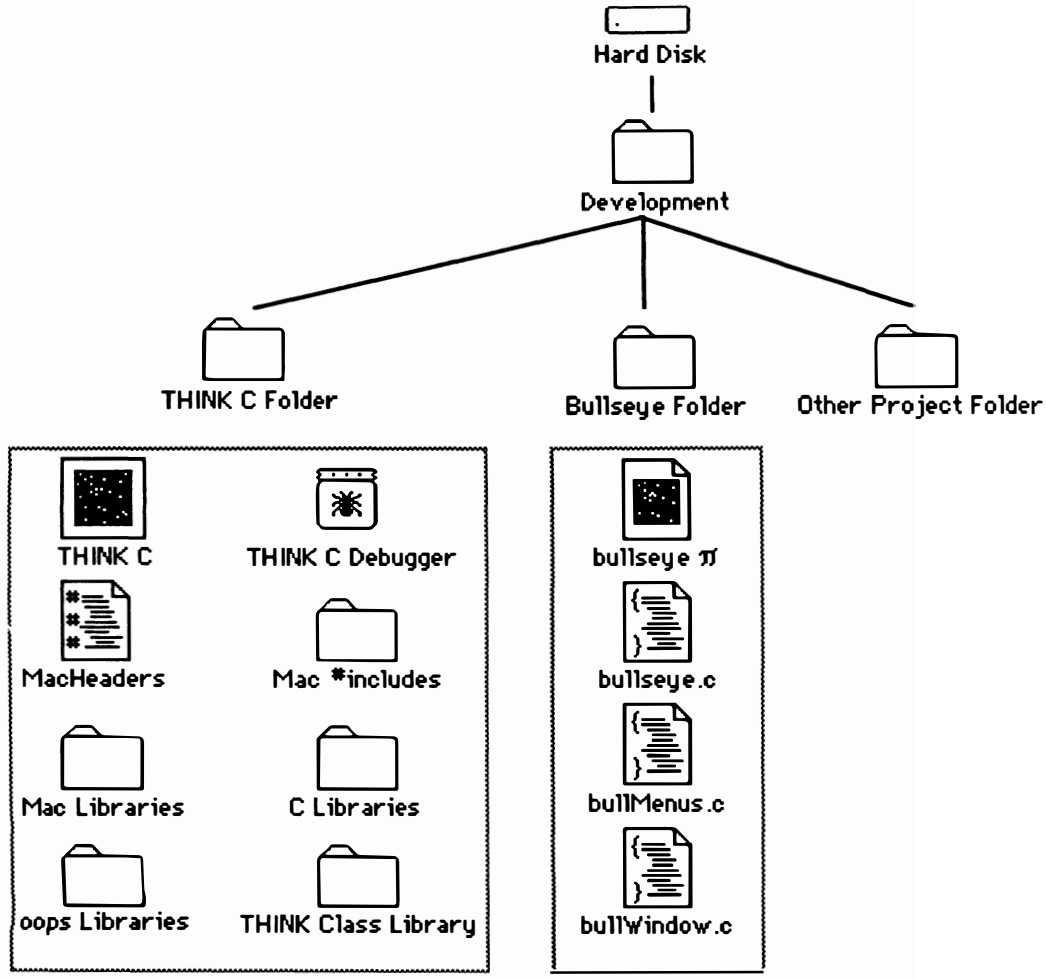
- `ANSI`
- `headers` folder

If space is tight on your floppy, remove the header files you won't use from the `headers` folder.

In the remaining space, you can copy over any other THINK C files and libraries you'll need.

Disk Layout Diagram

The following diagram shows the recommended disk layout. You don't have to set up your disk this way, but the important thing to remember is that your project folders should not be in the THINK C folder.



THINK C

PART TWO

Learning THINK C

3 Tutorial: Hello World
4 Tutorial: MiniEdit
5 Tutorial: Bullseye

Tutorial: Hello World
3

This chapter shows you how to put together an application with THINK C. The idea here is not to write a fancy program, but to show you how to build an application in THINK C. The program just writes the words "hello world" in a window on the screen.

Before you begin

Be sure you followed the instructions in Chapter 2 to put THINK C on your disk. If you gave your folders different names, your names may not match the ones in the pictures. It's all right as long as you know where your files are.

What you should know

You should know how to use the standard file dialog boxes to move around to different folders. If you don't know how to do this, read the user's manual that came with your Macintosh.

Topics covered in this chapter:

- Creating a project
- Creating the source file
- Compiling the source file
- Adding the libraries
- Running the project
- Creating an application

Creating the Project

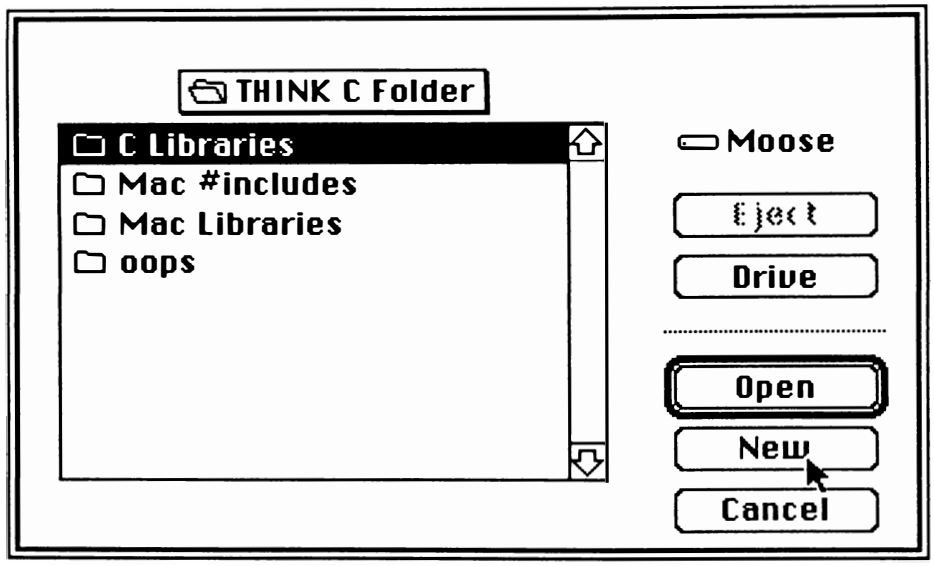
The first thing you need to do is create a folder for your project in the `Development` folder. This is the folder you'll use for all your development work.

Create a folder called `Hello Folder` in the `Development` folder. Do this now, before you start THINK C. You can use a different name if you like, but remember that your dialog boxes won't match the pictures in this chapter.

Generally speaking, you'll have a folder for each project you work on. The folder should contain your source files, your #include files, and the application's resource file.

When you've created the `Hello Folder`, open the `THINK C Folder` (the one that contains the THINK C application) and double click on the THINK C icon.

You'll see a dialog box that asks you to open a project.



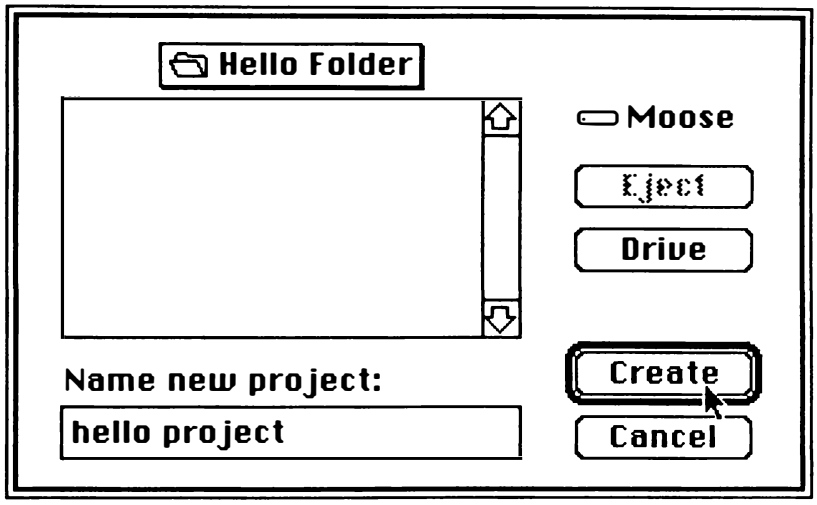
Since you're creating a new project, click on the New button.

You'll see another dialog box, one that lets you create projects.

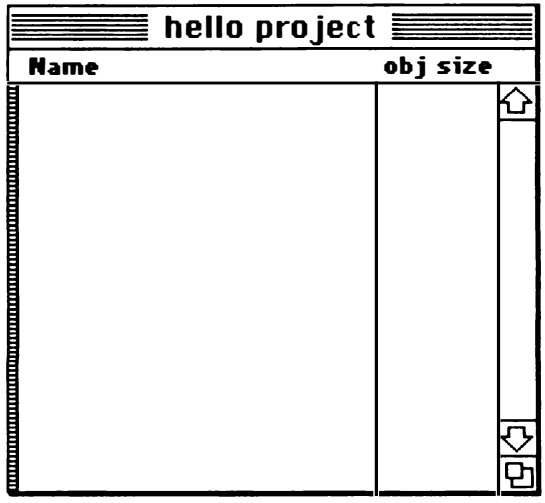
Move back to the `Hello Folder` you just created.

> **Note:** It's very important that you move to the `Hello Folder`.

Name the project `hello project`, and click on the Create button.



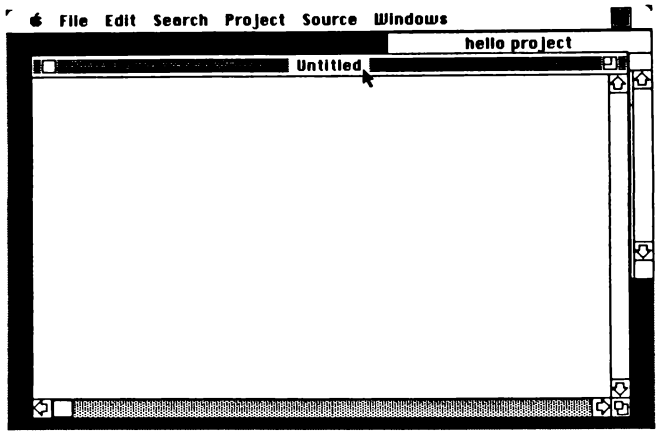
THINK C creates a new project document on disk and displays a project window:



The **Name** column shows the names of all the source files and libraries in your project, and the **obj size** column displays their sizes in bytes.

Creating the Source File

Now you're ready to create your source file. Choose **New** from the **File** menu to bring up an empty editing window.



Type this program into the editor window (you don't need to type in the comments if you're in a hurry):

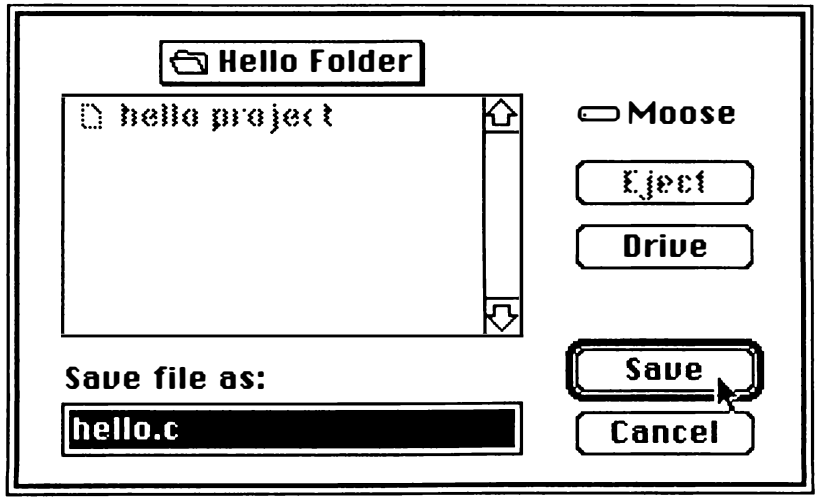
```
/*****
 * hello.c
 *
 * The hello world program for THINK C
 *
 *****/

#include <stdio.h>

main()
{
    printf("hello world\n");
}
```

The THINK C text editor works like most other text editors on the Macintosh. You can drag to select a range of text or double click to select words. Triple click to select an entire line. If you have a keyboard with arrow keys, you can use them to move around your file.

The text editor does not wrap text back to the left edge of the window when you type past the right edge of the window. Use the horizontal scroll bar at the bottom of the window to see any text that goes past the right edge.

For more information about the THINK C text editor, see Chapter 8.

When you've typed in the program, select **Save As...** from the **File** menu to save it. You'll get a dialog box like the one below. Name the file `hello.c`, and click on the Save button.

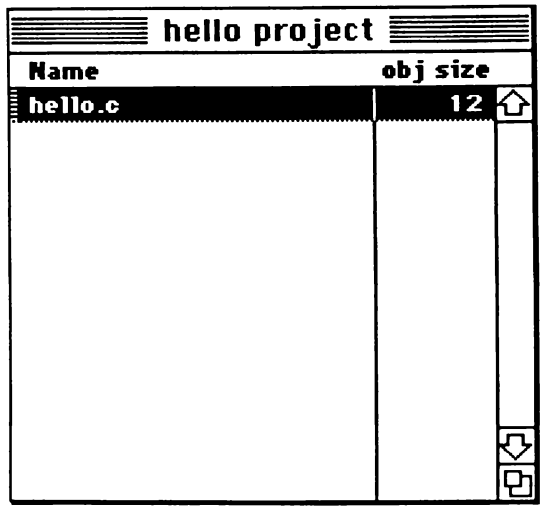


THINK C will only compile files that end in `.c`, but you can edit any text file with the THINK C editor.

Compiling the Source File

Now you're ready to compile your source file. Select **Compile** from the **Source** menu. THINK C displays a dialog box that shows how many lines have been compiled.

When THINK C compiles a source file, it adds its name and size to the project window. Your project window should now look like this:



THINK C keeps all the object code for your source files in the project document.

Did you get an error?

If you made a mistake typing the program, THINK C will display an error message in a dialog box. The message may say "syntax error." In this small program, about the only syntax error you can make is forgetting a quote, a parenthesis, or a semicolon.

Click anywhere in the dialog box to get rid of it. THINK C puts the insertion point in the line with the error. Look over your program to make sure everything is correct. Then select **Compile** from the **Source** menu.

If you get an error message that says "can't open #include'd file" like this one:

it means that THINK C wasn't able to find the #include file `stdio.h`. THINK C can't find the #include files if you didn't move the Libraries folder into the `THINK C Folder`. The best thing to do now is to start over from the beginning.

Quit THINK C and move the `Hello Folder` to the Trash. Then look in Chapter 2 to make sure you installed THINK C correctly. Once you're sure everything is OK, start again from the beginning of this chapter.

> **Note:** Throw the `Hello Folder` into the Trash only if you're starting all over. If you didn't get the "can't open #include'd file" error message, go on.

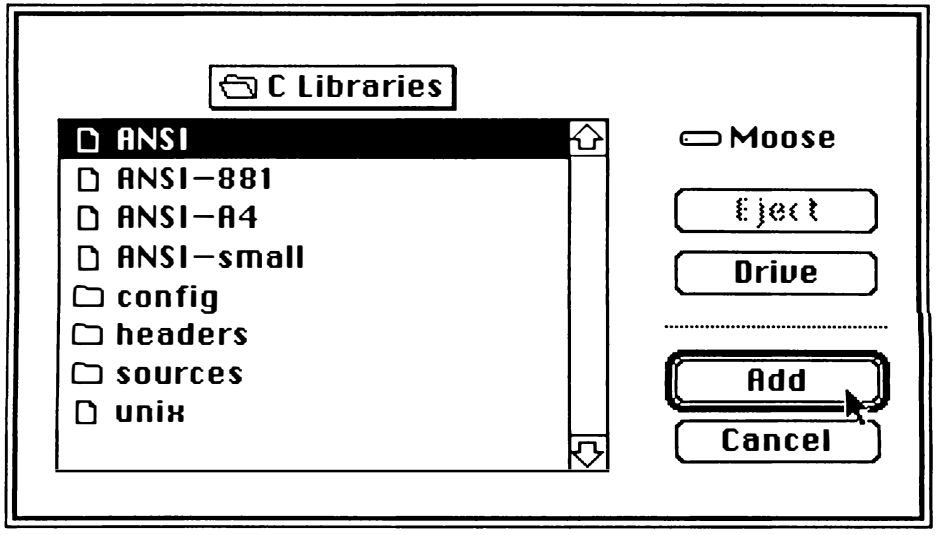
Adding the Library

If you were to try to run your program now, you'd get linking errors because the project doesn't know where the `printf()` function is defined.

The routine `printf()` is a standard C input/output function defined in the `ANSI` library. This library contains all the standard C library routines. To learn more about the routines in the `ANSI` library, see the *Standard Libraries Reference.*

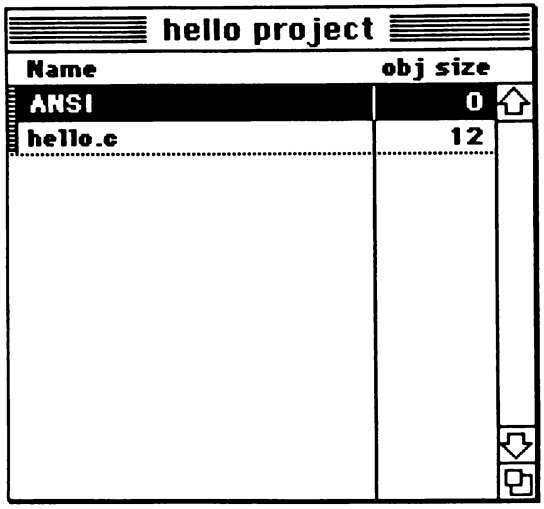
Next, you need to add the `ANSI` library to your project. To add the `ANSI` library, choose **Add...** from the **Source** menu.

When you get the standard file dialog box, open the folder called `C Libraries`. This folder contains all the libraries for ANSI and Unix compatibility, including the `ANSI` library. Select `ANSI`, and click on the Add button.



THINK C adds the name `ANSI` to the project window and then puts up the standard file dialog box again. `ANSI` is the only library you need for this project, so go ahead and click on the Cancel button.

Your project window should look like this:



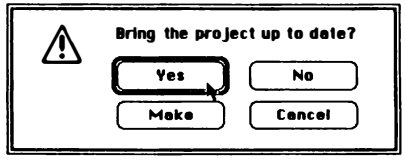
The object size for the ANSI library reads zero because when THINK C adds a library to your project, it doesn't load the code for it right away. This lets you add several libraries without waiting for them to load.

THINK C will load the library automatically when you run the project. Another way to load a library is to click on its name in the project window, and then choose **Load Library** from the **Source** menu. For this example, let THINK C load it for you.

Running the Project

Everything is all set to run the project. The source file is in the project window along with the libraries you'll be using. Now select **Run** from the **Project** menu.

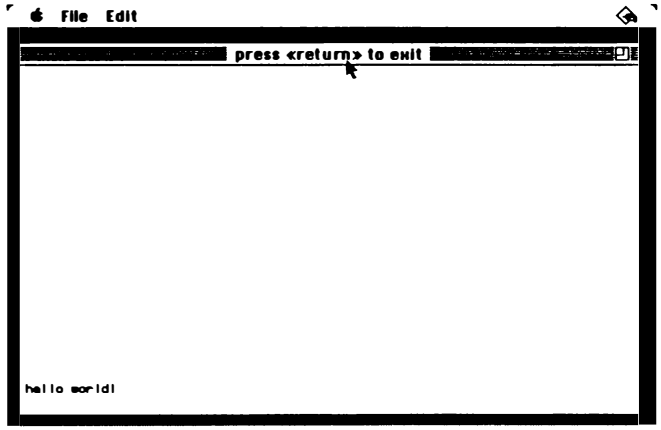
THINK C notices that the library needs to be loaded, so it puts up a dialog box asking you if you want to bring the project up to date:



Click on the Yes button. THINK C goes to disk to load the code for the ANSI library. It may take THINK C a little time to load it. Once it's loaded into the project, though, THINK C doesn't need to load it again.

Any time you choose to run your project and THINK C notices that you've made changes (added libraries or source files or edited source files) it will ask you if you want to update the project. If you say yes, it will compile the new or changed files and load the new libraries.

This program uses the `ANSI` library, so all output from `printf()` calls goes to a window called console. The console window emulates a generic terminal screen. You'll see the "hello world" string at the bottom of this window.

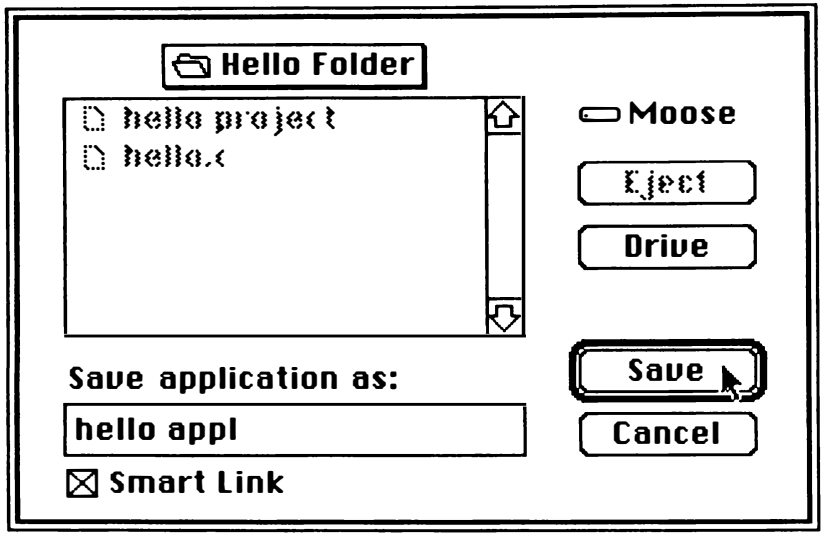


To exit the program, press the Return key or choose **Quit** from the **File** menu.

Creating the Application

As you develop a large application, you'll make changes to your source files. Each time you run your project, THINK C will recompile only those files that have changed. When you're ready to turn your project into a stand-alone double-clickable application, select **Build Application...** from the **Project** menu.

You'll see a dialog box asking you to name your application. Name it `hello appl`.



Leave the Smart Link box checked. This option tells THINK C to make your application as small as possible.

THINK C puts up a dialog box telling you it's linking your application. When it's finished, the application will be in the folder you chose.

If you're running without MultiFinder, quit THINK C to run your application. Use the **Quit** command in the **File** menu. If you're using MultiFinder, you don't need to quit first. Just bring up the window with the folder your application is in. Double click on your application and watch it run. That's all there is to it.

Where to Go Next

The tutorial in the next chapter is a more elaborate example of building an application with THINK C. It describes how THINK C reports errors when you compile and link, and it will show you some advanced features of the THINK C editor.

If you would rather explore on your own, read the chapters of the "Using THINK C" section that interest you. Or if you want to learn how to use THINK C's source level debugger, now, go to Chapter 5 and follow the tutorial there.

Tutorial: MiniEdit 4

Introduction

This chapter shows you how to use the more advanced features of THINK C. You'll build a small text editor based on the sample application described in Chapter 1 of *Inside Macintosh I*. One of the source files has a small, intentional bug to show you how THINK C makes it easy to fix mistakes.

You'll learn how to create a project, how to fix mistakes, how to run a project, how to build an application, and how to use a resource file.

Before you begin

If you didn't follow the "hello world" example in the last chapter, read it now to get an idea of how THINK C works in general.

Copy the folder `MiniEdit Folder` from disk `THINK C 4` to your THINK C folder. This folder contains all the files you'll need to follow this example.

What you should know

You should know how to use the standard file dialog boxes to move around to different folders. If you don't know how to do this, read the documentation that came with your Macintosh.

You will need to know which folder contains the file `MacTraps`.

Topics covered in this chapter:

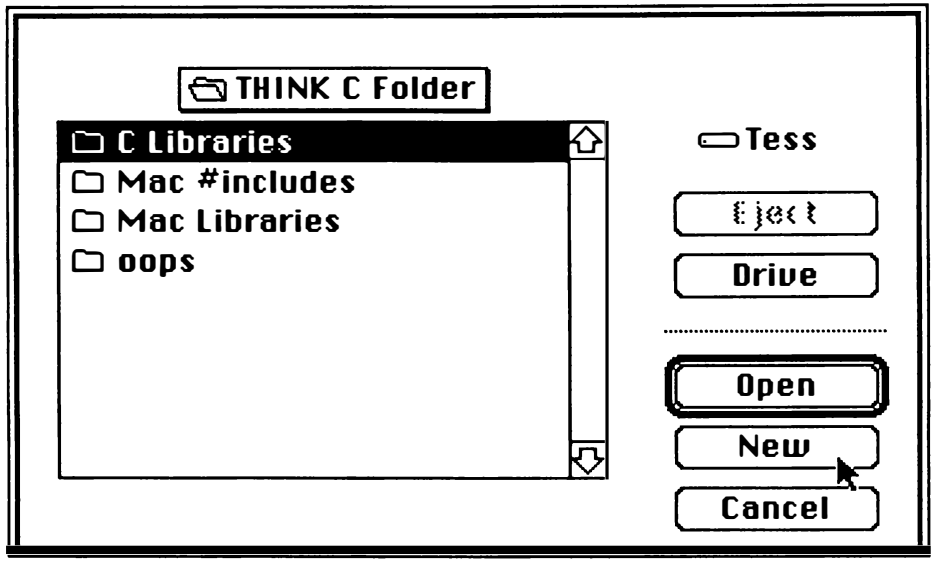
- Creating a project
- Adding the source files and libraries
- Compiling and running the project
- Fixing a bug
- Running the project again
- Creating an application
- Using a resource file
- Finishing up

Creating the Project

Make sure you copy the entire `MiniEdit Folder` from disk `THINK C 4`. This folder contains the source files you need to create the MiniEdit application as well as the application's resource file.

Generally speaking, you'll have a folder for each project you work on. The folder should contain your source files, your #include files, and the application's resource file.

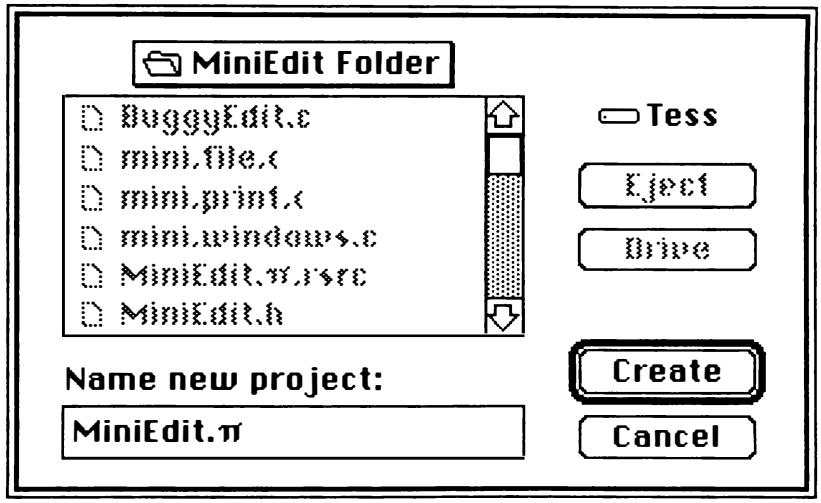
Open the THINK C Folder, and double click on the THINK C icon to begin. You'll see a dialog box that asks you to open a project.



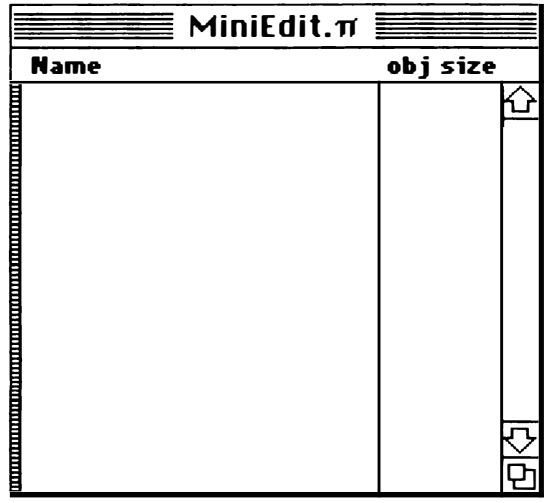
Since you're creating a new project, click on the New button.

When you get the next dialog box, move to the `MiniEdit Folder`, and name your project `MiniEdit.π`. Project names don't have to end in `.π`, though it's a good idea. For this example, it is important that you name your project `MiniEdit.π`. (To make a π, type Option-p.)

> **Note:** It's very important to move back to the `MiniEdit Folder`.



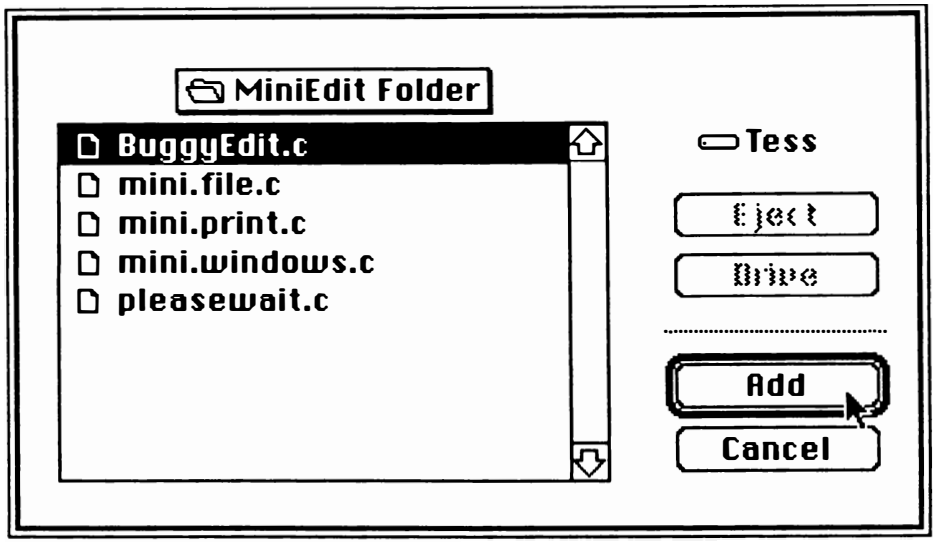
Click on the Create button. THINK C creates a project document on disk, and displays an empty project window.



Now you're ready to add the source files and the `MacTraps` library to your project. All the source files for `MiniEdit.π` are in the `MiniEdit Folder` you copied from disk `THINK C 4`.

Adding the Source Files and Libraries

Select **Add...** from the **Source** menu. You'll see a standard file dialog that lets you add source files and libraries.

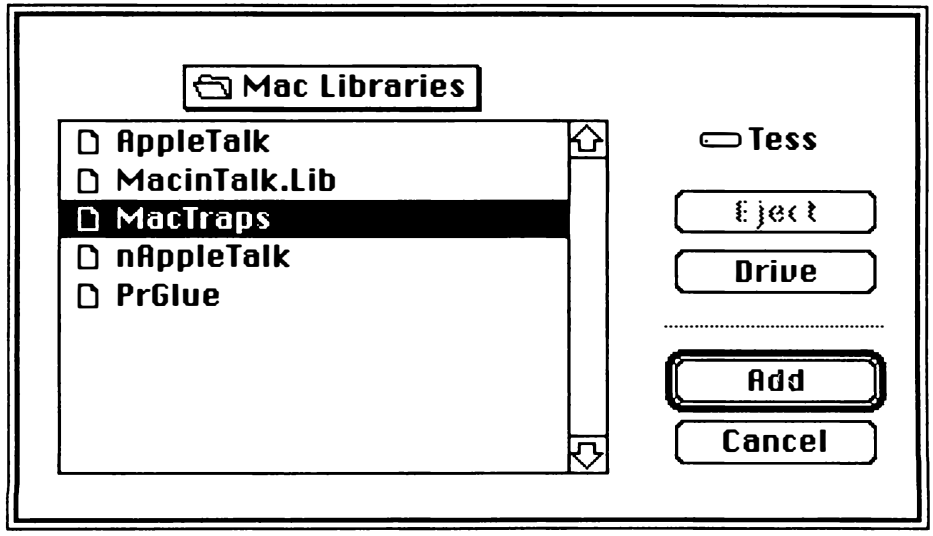


Double click on the first file in the file list, `BuggyEdit.c`. THINK C adds the file name to the project window and displays the standard file dialog again. Add all of the source files in the `MiniEdit Folder` to the project:

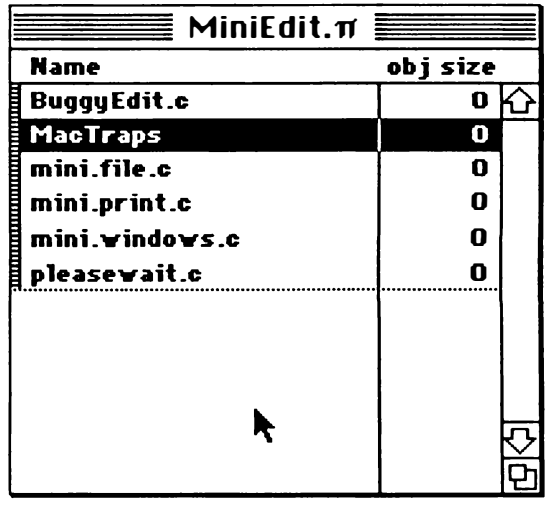
```
BuggyEdit.c
mini.file.c
mini.print.c
mini.windows.c
pleasewait.c
```

When you've added the last file, `pleasewait.c`, do not click on the Cancel button.

Move to the Mac Libraries folder, and add the `MacTraps` library.



All the files you need for this project are now in the project window. Click on the Cancel button now. Your project window should look like this:



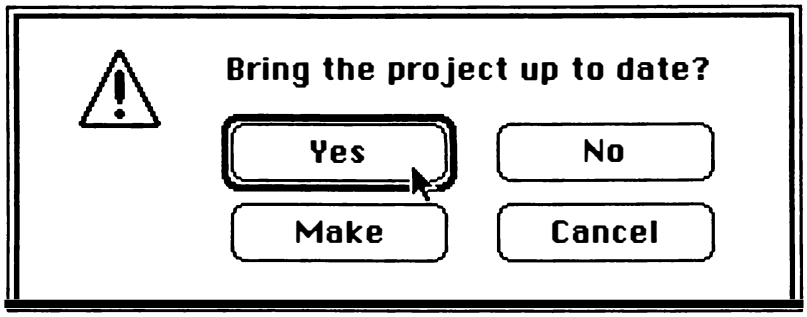
The **obj size** column displays the object size in bytes for each file. The sizes are all zero because you haven't compiled any files or loaded the `MacTraps` library.

Compiling and Running the Project

Before you can run your project, you need to compile the source files and load the `MacTraps` library. You can use the **Compile** and **Load Library** commands in the **Source** menu, or you can let THINK C take care of everything for you.

THINK C uses the project document to keep track of which files need to be compiled, so you can go ahead and run your project. Choose the **Run** command from the **Project** menu.

None of the files in the project have been compiled, so THINK C asks you if you want to bring the project up to date:



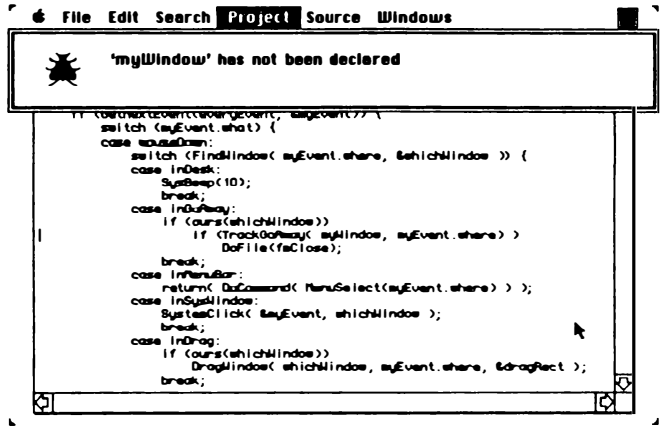
Click on the Yes button.

THINK C starts compiling the first file in the project. It displays a dialog box that shows how many lines have been compiled. (THINK C adds the number of lines in #includes files in the line count.)

In this example, THINK C doesn't get very far because `BuggyEdit.c` has a small intentional bug.

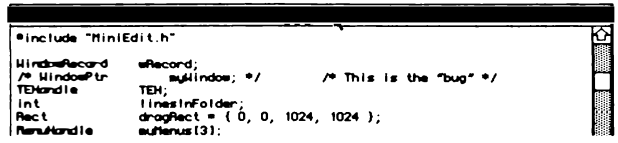
Fixing a Bug

When THINK C finds an error in your source file, it opens the file that contains the error and displays an error message in a dialog box. The insertion point is at the beginning of the line that contains the error. In this example, THINK C complains that a variable hasn't been defined.



To get rid of the dialog box, click anywhere in it or press the Return or Enter key.

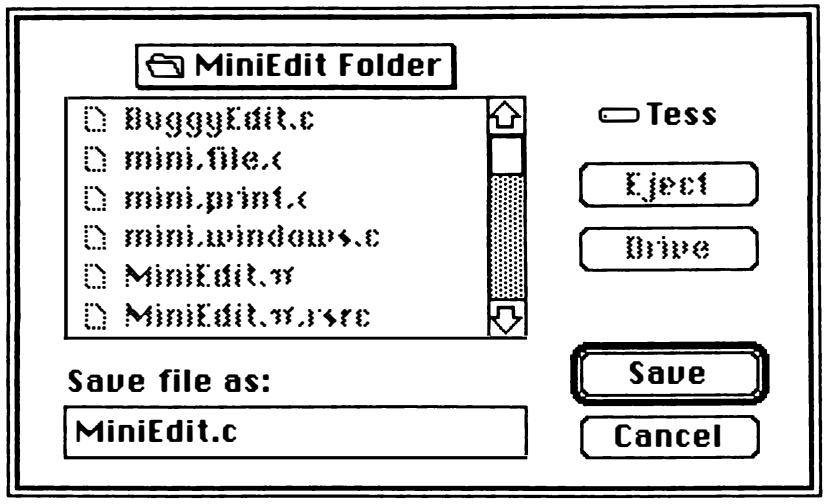
Scroll toward the beginning of the file, and you'll see that the declaration for `myWindow` is commented out:



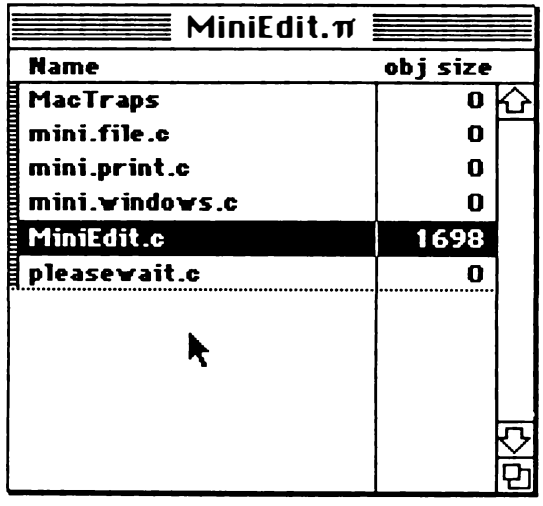
Remove the comments surrounding the declaration of `myWindow`.

Now, compile the file `BuggyEdit.c`. Choose the **Compile** command from the **Source** menu. THINK C will compile the source file without errors this time. Note that you don't have to save a file to recompile it.

Before you run the project again, save the changes you've made to `BuggyEdit.c`. Since the file no longer contains a bug, save it with a different name. Choose the **Save As...** command from the **File** menu, and save the corrected file as `MiniEdit.c`. (Make sure you're in the `MiniEdit Folder`.)



Now click on the project window. When you use the **Save As...** command on a file that is already in the project, THINK C changes the file's name in the project window as well. The file's object code is now associated with the new name.



To save a file with a different name without affecting the the project, use the **Save A Copy As...** command.

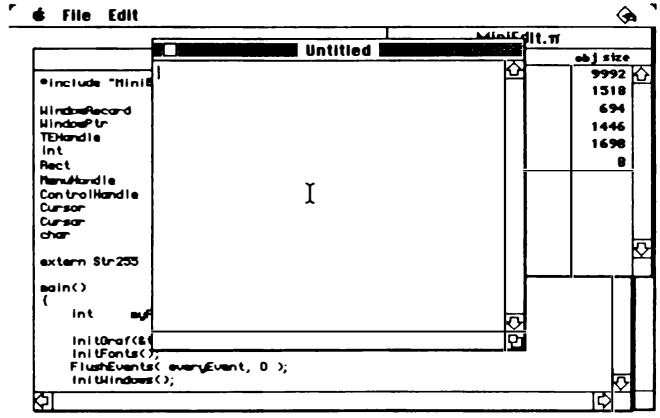
Now that you've fixed the bug, you can try running the project again.

Running the Project Again

Choose **Run** from the **Project** menu. When THINK C asks you if you want to bring the project up to date, click on the Yes button.

THINK C loads the `MacTraps` library, then it compiles all the files in the project. Since you already compiled `MiniEdit.c`, THINK C doesn't recompile it.

Once THINK C compiles the whole project, it launches it as if you had opened it from the Finder.

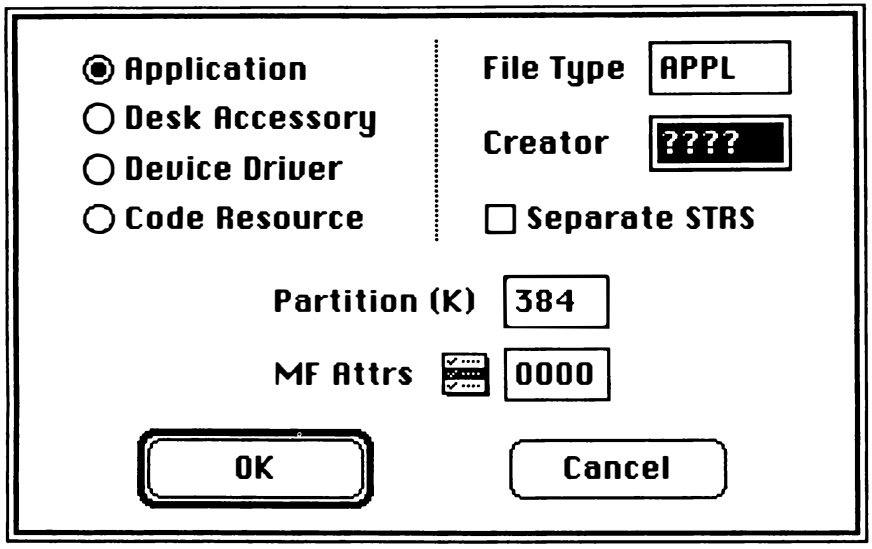


If you're using MultiFinder, THINK C launches your project in its own partition, so you can shift from your project back to THINK C. If you're not using MultiFinder, THINK C launches your project as if you had started it from the Finder. When you quit running your project, THINK C starts up again automatically.

Play with the MiniEdit application for a while if you like. You might want to make some changes. When you're satisfied with how the project runs, you're ready to turn it into a double-clickable application.

Building the Application

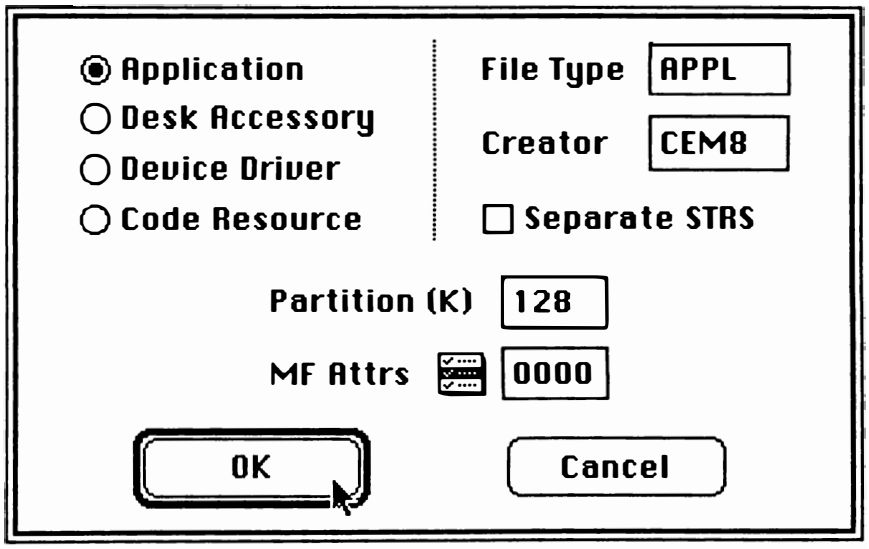
Turning your THINK C project into an application is easy. Choose the **Set Project Type...** command from the **Project** menu. You'll see this dialog box:



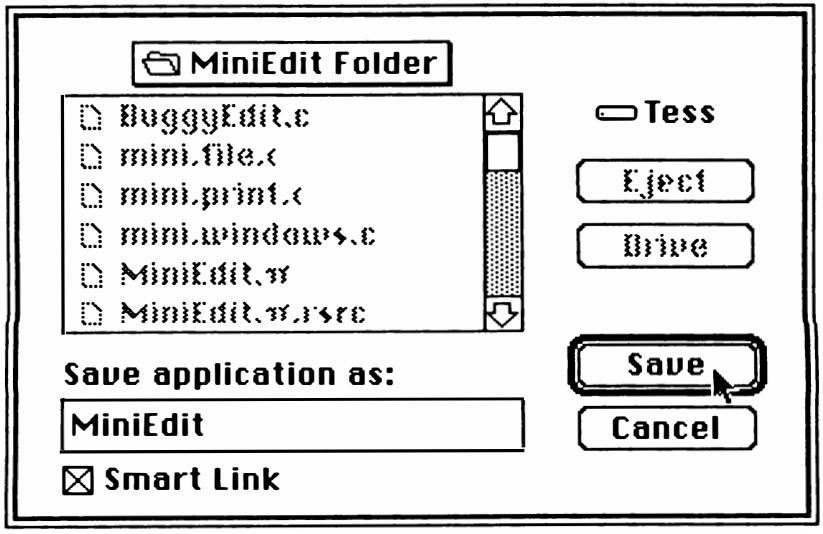
Set the creator to CEM8. This will ensure that your application will have the right icon when you build it. (CEM8 doesn't stand for anything. It's just unlikely that any other application on your disk has that signature.)

Set the partition size to 128K. Since MiniEdit is such as small program, it doesn't need the default 384K partition size. MultiFinder uses the partition size to determine how much memory to give to your application.

When the dialog box looks like this, click on the OK button:



Choose **Build Application...** from the **Project** menu. You'll see a dialog box like this:



Name the application `MiniEdit`. Leave the Smart Link box checked. This option tells THINK C to make the application as small as possible.

As it's building the application, THINK C gives you status messages. First it links all the object code. Then it copies the resource file for the project into the application. (The next section tells you how to use a resource file with a project.) When it's finished, you'll have a new application in the `MiniEdit Folder`.

MiniEdit

Using a Resource File

The `MiniEdit Folder` you copied from disk `THINK C 4` contains a file called `MiniEdit.π.rsrc`. This file contains the resources that the MiniEdit project uses.

When THINK C runs your project, it looks for a file named *`projectname`*`.rsrc`. (That is, the name of your project plus the characters `.rsrc` appended to it.) This file should contain the resources (menus, alerts, dialogs, etc.) that your project uses.

To create a resource file, you can use Apple's RMaker or ResEdit utilities (they're included in your THINK C package). `MiniEdit.π.rsrc` was created with ResEdit, so there's no RMaker source file for it.

Finishing Up

When you're finished working on a project, you can either close the project or quit THINK C. To close the project, use the **Close Project** command in the **Project** menu. When you close a project, THINK C displays a dialog box that lets you open or create projects.

To quit THINK C, choose **Quit** from the **File** menu.

Where to Go Next

The tutorial in the next chapter shows you how to use THINK C's source level debugger. It shows you how to activate the debugger, how to trace through your code, and how to examine and change the values of your variables.

If you feel comfortable with what you know so far, you might want to start creating your own applications right away. Use the next part of the manual, "Using THINK C", when you need help on a particular topic. You'll probably find Chapter 8, "The Editor" useful now.

Tutorial: Bullseye 5

Introduction

This chapter shows you how to use THINK C's source level debugger. You'll use an example program called Bullseye, which is included in your THINK C package. Bullseye is a simple application. It draws a series of concentric circles in a small window. Bullseye's **Width** menu lets you select how wide each of the rings is.

Before you begin

Make sure you're running MultiFinder on a Macintosh with at least 2Mb of memory.

Copy the `Bullseye Folder` from disk `THINK C 4` to your development folder. The `Bullseye Folder` contains all the files you'll need for this example.

Make sure that the file `THINK C Debugger` is in the same folder as THINK C (the `THINK C` folder). This file must be named `THINK C Debugger`.

What you should know

Before you try this example, you should know how THINK C works. You should know how to open a project, how to edit source files, and how to run a project. If you're not familiar with any of these operations, go back and read (or try) the examples in the last two chapters.

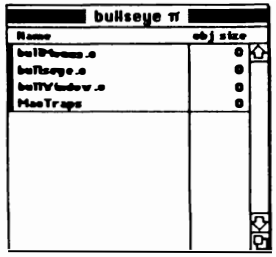
Topics covered in this chapter:

- Opening the Bullseye project
- Turning the debugger on
- Watching the program run
- Examining and setting variables
- Examining structs and arrays
- Expressions and contexts
- Quitting the debugger

Opening the Bullseye Project

If you're at the Finder, double click on the `Bullseye.π` project in the `Bullseye Folder`. If you're already in THINK C, use the **Open Project...** command in the **Project** menu to open the `Bullseye.π` project.

Bullseye consists of three source files and the `MacTraps` library.

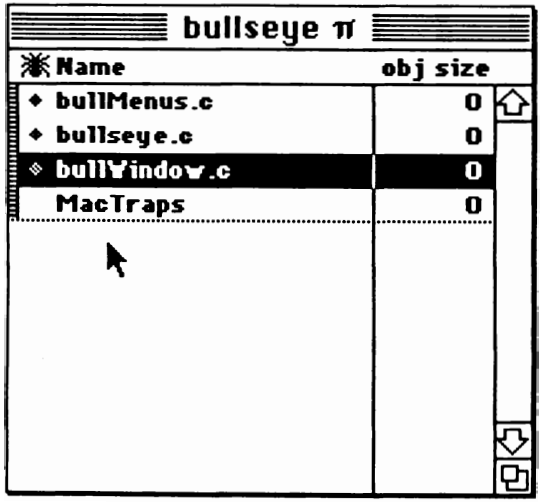


Note that none of the files have been compiled, and that the `MacTraps` library hasn't been loaded.

Turning the Debugger On

THINK C ordinarily runs your project without the debugger. Choose the **Use Debugger** command from the **Project** menu.

When the source debugger is on, THINK C adds a "bug" column in the project window to the left of the Name column.



Generating the debugging tables

The gray diamonds in the "bug" column let you know that THINK C will generate special debugging tables for a source file. These tables go into your project document along with the source files' object code. THINK C never generates additional code when you run the debugger.

Running the project

Choose **Run** from the **Project** menu to let THINK C compile and load all the files in your project. THINK C will generate debugging tables for all the files as well.

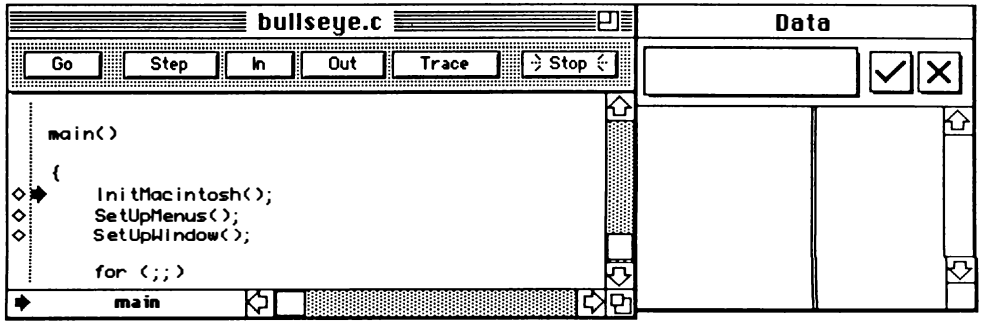
Watching the Program Run

Instead of running your project, THINK C launches the source debugger, which controls the execution of your program. The debugger displays two windows at the bottom of your screen. If you're using two screens, the debugger windows are on the second screen instead.

The window on the left is the Source window. It contains the source text of your program. The window on the right is the Data window. You use this window to examine and set the values of your variables as you debug your program.

The Source window

The Source window shows you the source of your program. The title of the window is the name of the source file you're looking at.



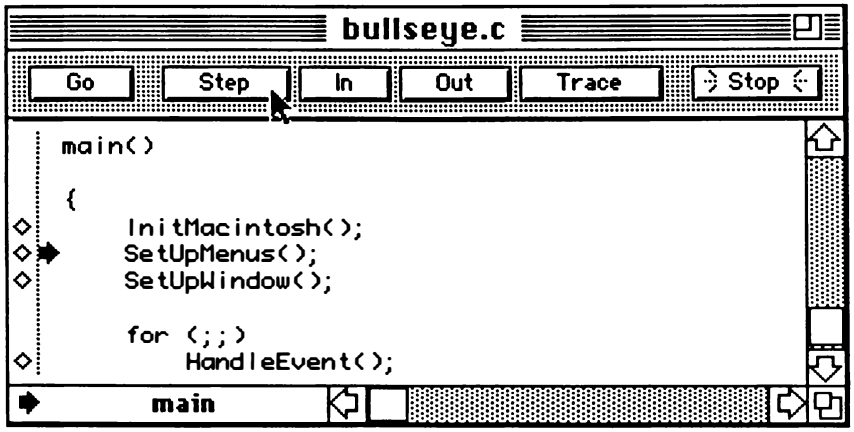
Along the top of the Source window is a row of six push buttons called the **status panel**. These buttons control the execution of your program. The status panel also shows you the state of your program. Right now, the Stop button is lit to show you that your program is stopped.

The arrow to the left of the first line of the program points to the statement that's about to be executed. This is called the **current statement**.

The hollow diamonds at the left of the Source window are **statement markers**. The debugger displays a statement marker for each statement in your program. (Loosely speaking, a statement is a line that generates code.) Later, you'll use the statement markers to set breakpoints.

Stepping through statements

Click on the Step button in the status panel.



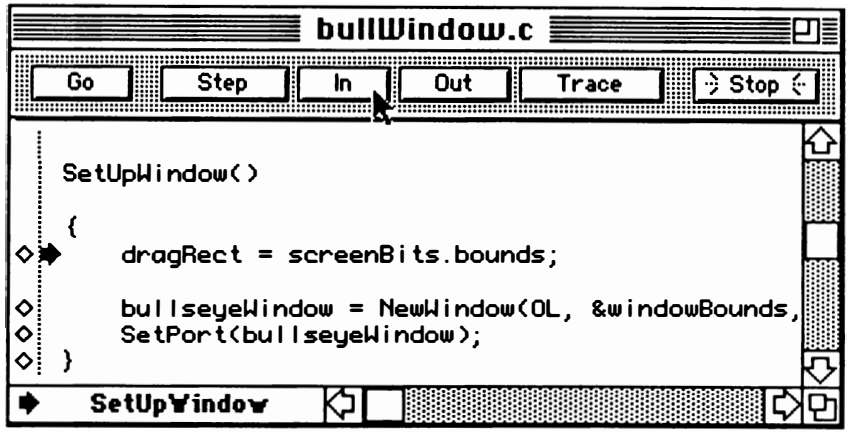
The Step button lights up for a moment, the current statement arrow moves to the second statement, and the program stops again.

The Step button lets you execute your program line by line. You can use the **Step** command in the **Debug** menu or type Command-S to do the same thing.

Press the Step button (or type Command-S) one more time so the current statement arrow points to the call to `SetUpWindow()`. This function creates the window that Bullseye uses.

Stepping into functions

To see how `SetUpWindow()` works, press the In button on the status panel (or type Command-I). (This command is called **Step In** in the **Debugger** menu.)



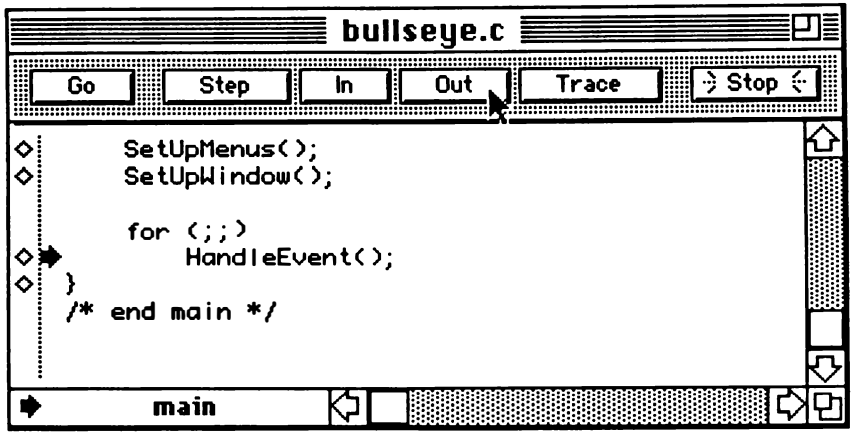
Now the current statement arrow points to the first line of the `SetUpWindow()` function. This function is in the file `bullWindow.c`, so the title of the window changes to let you know what file it's displaying.

> **Note:** The current statement arrow doesn't have to be right before a function call for the In button to work. The **Step In** command executes every statement until the program counter is no longer in the current function. Another way to think of the **Step In** command is: "Keep going until you fall into a function." (**Step In** also stops execution if you fall out of the current function.)

Stepping out of functions

You can see the entire `SetUpWindow()` function in the source window. It's a pretty straightforward function, and you can rest assured it works.

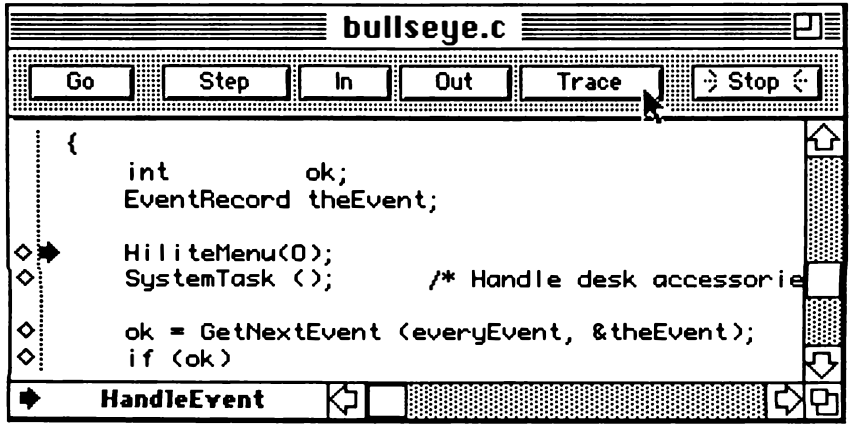
Click on the Out button to leave the `SetUpWindow()` function. The Source window now shows that the debugger is ready to execute the function `HandleEvent()` in the `bullseye.c` file. (Pressing the Out button is the same as choosing **Step Out** from the **Debug** menu.)



The Out button steps through each statement in the current function until the current statement arrow leaves the function.

Tracing every statement

Now click on the Trace button (or type Command-T). The current statement arrow now points to the first statement of the `HandleEvent()` function.



Tracing takes you to the next statement even if it has to step into a function. If you were to continue tracing, you'd stop at every statement. Stepping, on the other hand, *never* dives into a function.

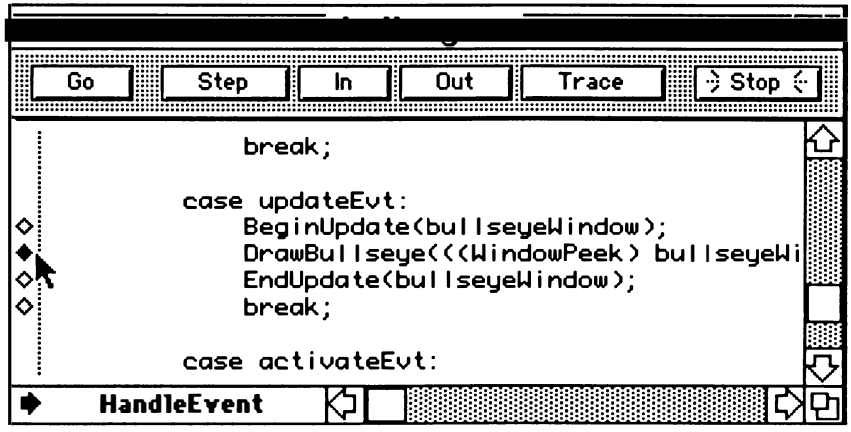
> **Note:** The In button actually does a Trace until the current statement arrow leaves the current function.

Setting a breakpoint

Since the `SetUpWindow()` function opened a new window, the program will get an activate event the first time through the event loop. In Bullseye, all the program does on activate events is call `InvalRect()` on the whole window, so the second time through the event loop it gets an update event.

You could Step or Trace to verify that this is what really happens. A faster way is to set a breakpoint at the function that redraws the window.

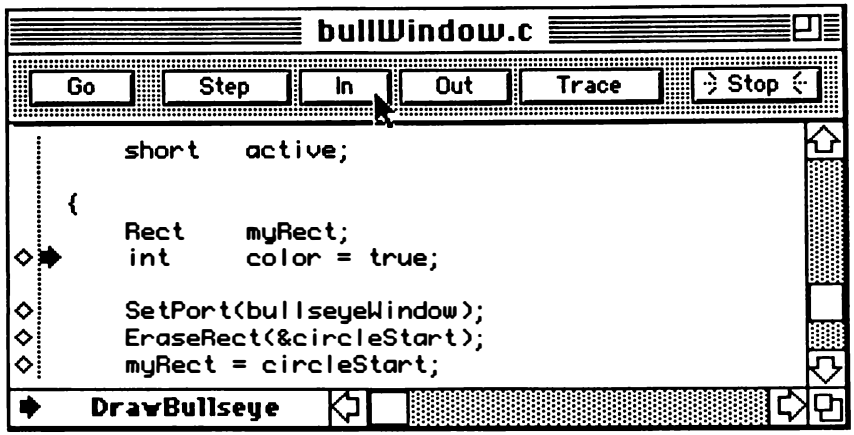
Scroll down in the source window until you get to the code that handles update events. Click on the statement marker to the left of the `DrawBullseye()` function.



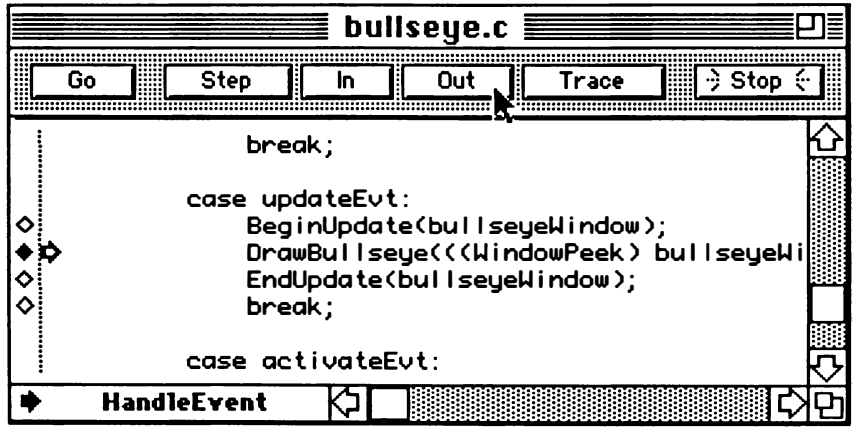
The hollow diamond turns black to indicate that you've set a breakpoint. You can set as many breakpoints as you like this way. When your program is about to execute a statement that has a breakpoint, it will stop. To remove a breakpoint, just click in the filled diamond.

To start your program running, press the Go button. The program runs for a few moments and then stops. The current statement arrow is at your breakpoint.

Press the In button to step into the `DrawBullseye()` function. (This function is in the `bullWindow.c` file, so that's the file you see in the source window now.)



Click on the Step button to watch how the program draws a bullseye in the window. If you get bored, press the Out button. Whether you Step or Step Out, you'll eventually end back at the call to `DrawBullseye()`.



Note that the current statement arrow is hollow. This means that there are still some instructions left to execute in the statement. You'll see hollow arrows when the statement is making an assignment or cleaning up the stack after stepping out of a function.

Before you go on, clear the breakpoint. Just click on the filled diamond.

Letting the program run

Press the Go button to let the program run. You can set and clear breakpoints while your program is running.

When you click in the Source window to set breakpoints, your application will go to the background, and the debugger comes to the foreground. If you press the Go button when your program is running, the debugger brings it to the foreground.

Stopping the program

To stop your program, click on the Stop button or, if the debugger is the frontmost application, press Command-Period. Your program will stop as it's coming out of one of the event-fetching routines (`GetNextEvent()` or `WaitNextEvent()`).

If your program is stuck in a loop or if you want to stop it without waiting for control to return to the event loop, you can use the **panic button**, Command-Shift-Period, to stop your program. Be careful when you do this, though, because the debugger will stop execution no matter what it's doing.

Viewing other files

Sometimes you'll want the Source window to display another file in your project. For example, you might want to set a breakpoint in a file other than the one the current statement arrow is in.

To see a different file in the Source window, first bring THINK C to the foreground. You can do this several ways: type Command-0, choose you project name from the **Windows** menu, click on the project window, or choose THINK C from the list of applications at the bottom of the **Apple** menu.

Next, click on the file name in the project window. Then choose **Debug** from the **Source** menu. The source file will appear in the Source window, and you can set breakpoints in it.

To display the file that contains the current statement again, just click on the current function name at the bottom left of the Source window.

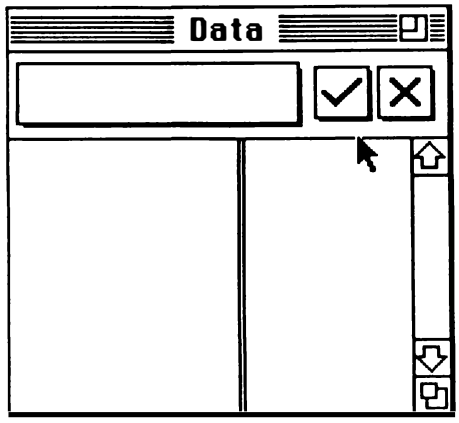
Examining and setting variables

Tracing your program's execution lets you see what your program is doing. But to really fix bugs, you need to be able to examine your variables. That's what the Data window is for.

If you've quit the Bullseye program, start it up again. Select **Run** from the **Project** menu, and when you see the debugger window, press the Go button in the status panel.

The Data window

The Data window appears to the right of the Source window. The best way to think about this window is to treat it as a spreadsheet.



At the top of the Data window is the **entry field** and two push buttons. The button marked with a check mark is the **enter button**, and the button marked with an X is the **deselect button**.

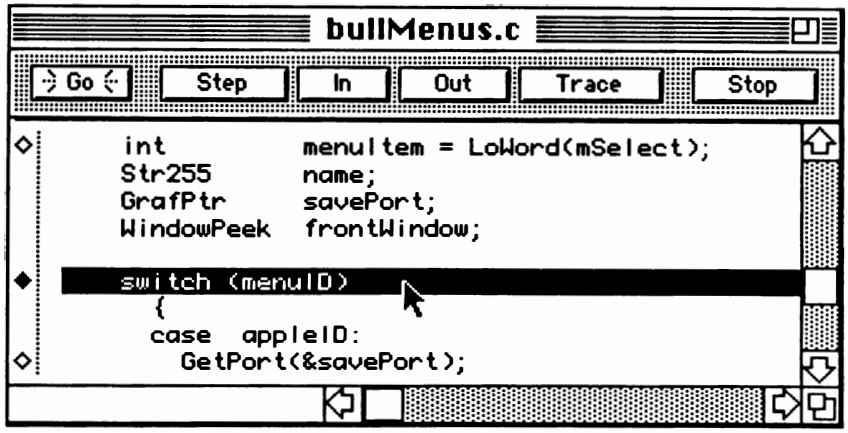
Below the entry field and the buttons are two columns. The column on the left is the **expression column**, and the one on the right is the **value column**. You enter expressions (usually variable names) in the left column, and their values appear in the right column.

Examining variables

Suppose you want to watch the value of the `menuID` variable in the `HandleMenu()` function. First, make sure the `bullMenus.c` file is displayed in the source window. If it's not, bring the project window to the front, click on the name `bullMenus.c`, and select **Debug** from the **Source** menu.

Next, scroll down until you see the `HandleMenu()` function, and set a breakpoint at the `switch` statement. Remember that you can set breakpoints even while your program is running.

After you set the breakpoint, click once on the line that contains the switch statement to select it.

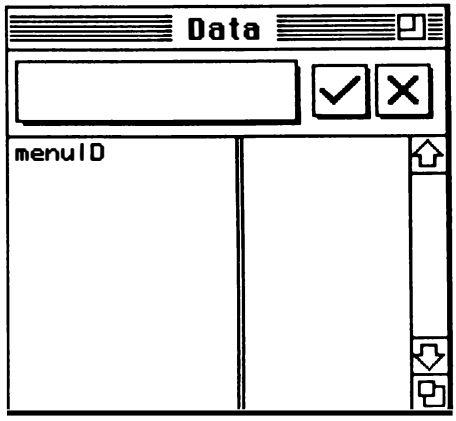


You select a line to give the debugger a **context** for evaluating `menuID`. In this case, you're saying that you want to know the value of `menuID` right before the `switch` statement.

Expressions in the Data window have either **local scope** or **global scope**. An expression has local scope if it refers to variables with dynamic storage; in other words if it refers to non-static variables local to a function. All other expressions have global scope.

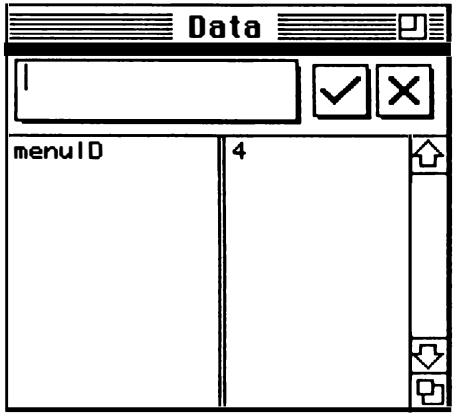
Click in the Data window. You'll see the insertion point blinking in the entry field. Type `menuID` in the entry field and press the Return key.

The debugger compiles the expression (it takes about a second) in the context of the selected line. Right now, the Data window doesn't show a value for `menuID` because the program isn't stopped there.



Now, go back to the Bullseye program. Click on the Bullseye window, click on the Go push button, or type Command-G to bring Bullseye to the foreground.

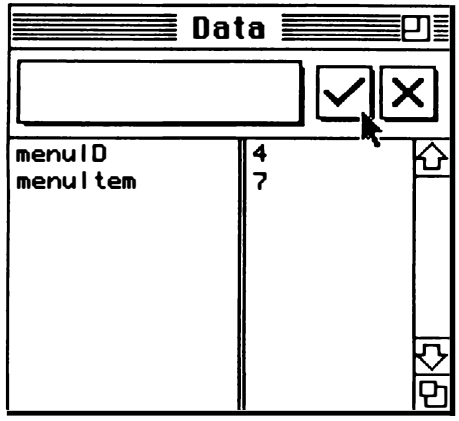
Next, select **7** from the **Width** menu. Your program stops at the breakpoint when you release the mouse button, and the value of `menuID` appears in the value column.



Any time your program stops, the source debugger displays the values of expressions that have global scope. Then it displays the values of expressions with local scope whose context is the same as the current function. Finally, the debugger clears the values of local expressions whose context is not the current function.

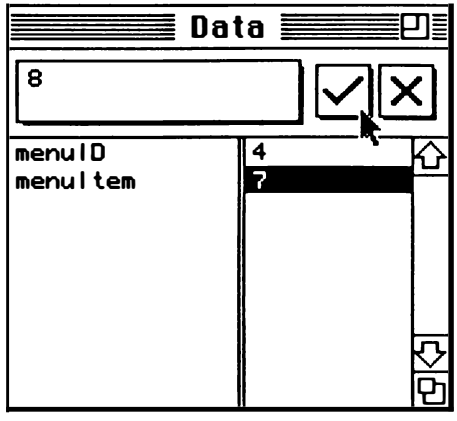
Changing the value of a variable

Click in the Data window again, and type `menuItem`. This variable contains the item number of the selected menu item. When you press the Return key, the source debugger shows you its value.



To change the value of a variable, click on its value and type a new one in the entry field. When you click on the enter button, the value of the variable changes. Here's an example.

Click on the value of `menuItem` (the right column) to select it. Its value appears in the entry field as well. Now type 8 as a new value for it. Click on the enter button to assign the new value to the variable.



When you click on the Go button, the Bullseye program behaves as if you had chosen **8** from the **Width** menu.

To remove an expression from the Data window, select it and choose **Clear** from the **Edit** menu or press the Clear key.

> **Note:** You can enter the same expression more than once in the Data window. You might want to do this to lock one of the expressions so you can compare it to the same expression later in the program. See "How and when the source debugger evaluates expressions" below.

Now choose the **Clear All Breakpoints** command in the **Source** menu to make sure there aren't any breakpoints set before you go on to the next section. Then click on the Go button to start the program running again.

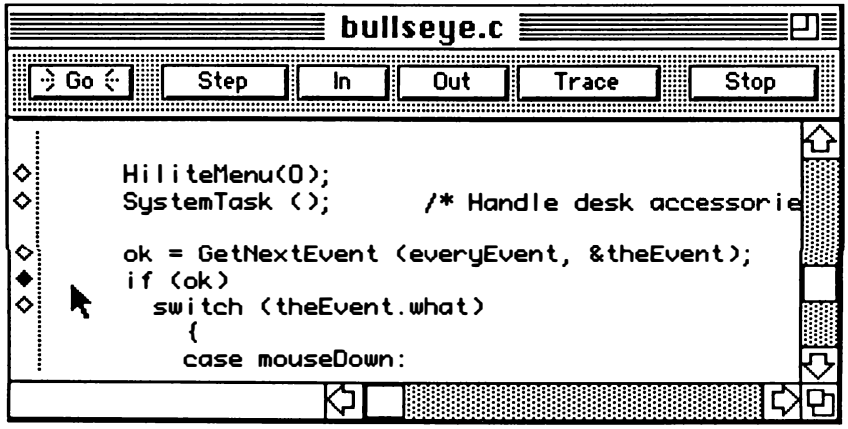
Examining structs and arrays

The data window lets you examine and modify structs and arrays, not just simple variables. When you display a struct or union in the data window, its value appears as `struct 0x000000` or `union 0x000000`. Arrays appear as `[] 0x000000`. (The real address appears instead of `0x000000`, of course.)

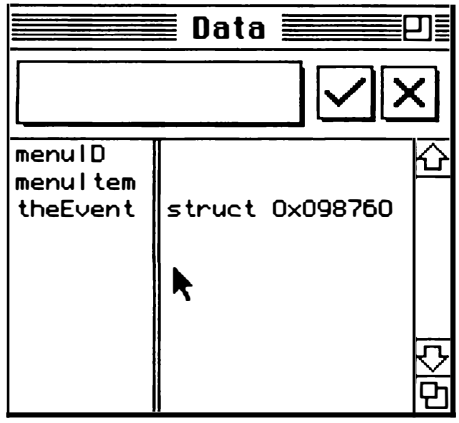
> **Note:** Anything you read here about structs applies to unions as well.

When you double click on one of these values, the debugger displays another window for the struct or array.

To see how this works, make sure the Bullseye program is still running. Display the file `bullseye.c` in the Source window, and set a breakpoint on the line right after the call to `GetNextEvent()` in the function `HandleEvent()`.



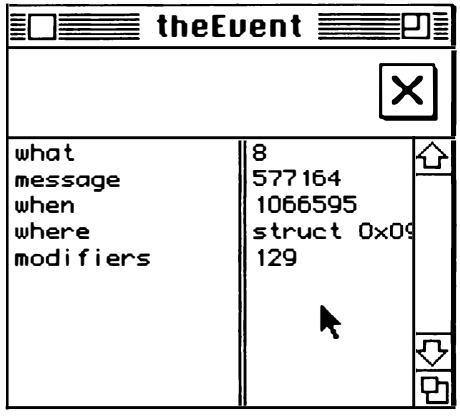
Now click in the Bullseye window. The program will stop at the breakpoint. When it does, type `theEvent` in the entry field of the Data window, and press the Return key.



The debugger displays the word `struct` and the address of the struct. If you can't see the entire value, click on the center separator bar and drag it to the left. Or you can make the window bigger.

> **Note:** If you don't select a line to give a variable a context, the debugger uses the current statement.

Double-click on the value of `theEvent`. The debugger displays a window. The names on the left are the fields of the struct.



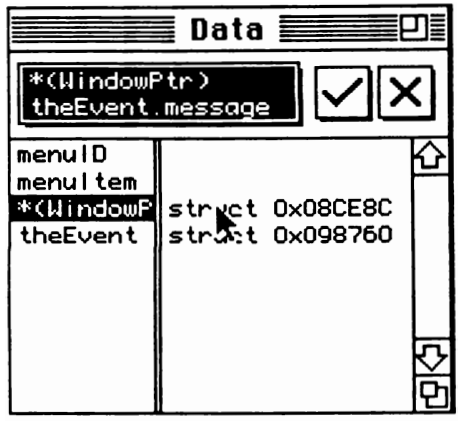
You can edit the values of the fields, but you can't edit the names.

The `what` field indicates that you're looking at an activate event (`activateEvt = 8`). In activate events, the `message` field points to the window record that gets the activate event.

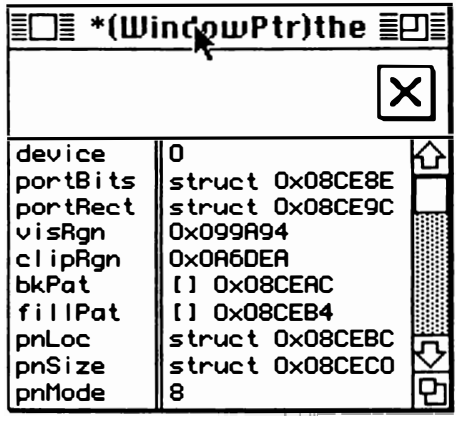
Double-click on the `message` field. The debugger enters a new expression in the main Data window: `theEvent.message`.

> **Note:** Double-clicking on the left column of any Data window creates a new entry in the main Data window.

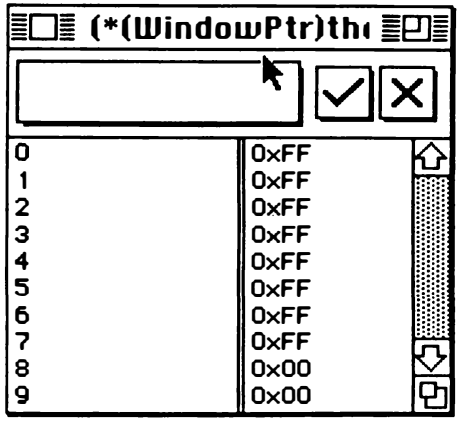
Edit the expression so it reads `*(WindowPtr)theEvent.message` so you can look at the `WindowPtr`.



Double-click on the value of the new expression. The debugger displays another struct window.



Scroll down to the pnPat field, and double-click. You'll see an array window.



Because C compilers don't enforce array bounds, array windows have "infinite" scroll bars. Unlike structs, you can select an index in the left column and change it. When you do so, the window shows the array from the index you entered.

When you double click on the value of a pointer variable, the debugger inserts a dereferenced expression in the Data window and displays its value. To see a pointer as an array, change its format to Address.

To get rid of a struct or array window, you can click on its close box, press the Clear key, or select **Clear** from the **Edit** menu. When you use the Clear key or the **Clear** command, the debugger removes the expression associated with the window from the main Data window. If you click on the window's close box, only the window goes away; the expression in the main Data window remains.

Expressions and contexts

You can type any C expression in the entry field of the Data window as long as it doesn't have any potential side effects. This means that you can't type in a function call, an assignment statement, or any expression that uses the autoincrement (++) or autodecrement (--) operators.

Every expression you type in the entry field is compiled in a context. The context is the selected line of the Source window. If no line is selected, the context is the line that the current statement arrow points to.

To see the context of an expression, click on the expression in the left column of the Data window, and select **Show Context** from the **Data** menu. The source debugger will display the context in the source window.

To change the context of an expression, click in the source window at the line you want to use as a context. Then select an expression in the Data window and choose **Set Context** from the **Data** menu. A shortcut is to hold down the Option key as you click on the enter button.

If you edit an expression, its context will be the context of the original expression. You can change its context by holding down the Option key as you click on the enter button as described above.

How and when the source debugger evaluates expressions

Expressions in the Data window have either local scope or global scope. An expression has local scope if it refers to variables with dynamic storage; in other words if it refers to non-static variables local to a function. All other expressions have global scope.

Every time your program stops, the debugger evaluates the expressions in the Data window. It displays the values for expressions with global scope and the values of expressions with local scope. Expressions that don't have a global or local context are cleared to make the window less cluttered.

If you want to make sure that the debugger doesn't redisplay a value, select it and choose **Lock** from the **Data** menu. A small lock icon appears next to the expression. This command is useful if you want to compare the value of the same expression at different times. You can also lock expressions to keep their values from being cleared when they go out of scope.

Display formats

The way the debugger displays expressions depends on their type. You can change the format with the formatting commands in the **Data** menu. To change a format, select an expression from the Data window. Then choose the format from the debugger's **Data** menu.

Not all formats are available for all types. Defaults are in italics:

Type	**Formats Available**
integers	*decimal*, hex, char
unsigned	*hex*, decimal, char
pointers	*pointer*, address, hex, C string, Pascal string
arrays	*address*, C string, Pascal string
structs	*address*
unions	*address*
functions	*address*
floats	*floating point*

This is what the display formats look like:

Format	Example
decimal	`4523345, -23576`
hex	`0xA09E1487`
char	`'c', 'TEXT'`
C string	`"abcdef\nghi\033"`
Pascal string	`"\pabcdef\nghi\033"`
pointer	`0x7A7000`
address	`[] 0x09FE44, struct 0x08FC14`
floating point	`1961.0102`

The C string and Pascal string formats display non-printing characters in backslash form. Whenever it can, the debugger uses the built-in escape characters (`\n`, `\r`, `\b`); otherwise it uses `\nnn`, where `nnn` is an octal value.

Of course, you can use type casting to use formats that aren't normally available. For example, if you really wanted to see an integer, `i`, as a C string, you would type this expression: `(char *) i`.

To see any pointer as an array, just change its format to Address. This way, when you double-click on its value, you'll see an array window instead of the value of what the pointer points to.

Quitting the Debugger

The best way to quit the debugger is to quit your application. You should use the **ExitToShell** command in the debugger's **Debug** menu only when you can't use your application's **Quit** command.

THINK C

PART THREE

Using THINK C

6 Overview
7 The Project
8 The Editor
9 Files & Folders
10 The Compiler
11 The Debugger
12 Assembly Language
13 Libraries

Overview 6

Introduction

This chapter describes the THINK C environment and how to write a program in THINK C. The rest of the chapters in this part of the manual discuss specific aspects of the components of the THINK C environment.

Topics covered in this chapter:

- The THINK C environment
- The Project
- Writing a program in THINK C
- Using THINK C

The THINK C Environment

THINK C is a complete integrated development environment, not just a C compiler for the Macintosh. Traditional development environments consist of three separate applications: the editor, the compiler, and the linker. It is up to you create your source files with a text editor, run each file through the compiler, and finally link all your object files.

In THINK C, the three components work in concert as parts of the same application. This way, THINK C knows when you've edited a file. The compiler produces object code that the linker can put together in an instant. Then THINK C can launch your program. And because THINK C is still running, it can launch the source level debugger so you can debug your program.

The Project

The project is at the heart of the THINK C development environment. What you see on the screen is a project window. It contains a list of all the files that comprise your program. Next to each file name is the size of that file's object code.

Rather than producing a separate binary object code file, THINK C keeps all the object code in the project document in ready-to-link form.

Because the project document knows all the files that make up your program (including header files), it can keep track of changes. When you edit a source file, the project manager

marks it for recompilation. When you edit an #include file, the project manager marks all the files that use it.

Writing a Program in THINK C

Writing a program in THINK C is like writing a program in any other development environment. You create your source files, compile them, then link the object code to create an executable file. The difference is that in THINK C, you use the same application to do all of this.

Creating source files

When you write a program in THINK C, you create a project document. Usually, the project document is in a folder that you'll use for all the files related to your program. Next you'll create your source files and add your libraries. THINK C source files are standard text files, so you'll be able to use existing source files. The THINK C editor provides some features that help you edit C source code. Its search facilities include a pattern matching option based on Grep, and a multi-file search that looks for strings in any file in your project.

Adding libraries

Virtually every program you write will need to access the Macintosh Toolbox. You can call any Macintosh Toolbox routine exactly as it's described in *Inside Macintosh*. The code for Toolbox routines marked [Not In ROM] as well as the glue code needed to call some of the other Toolbox routines is in the `MacTraps` library.

Your THINK C package also includes several other libraries you can use in your programs. The `ANSI` library contains the standard ANSI functions defined in the ANSI standard. The `unix` library contains UNIX system functions including memory calls. You can use these libraries when you port code from other systems. You can also create your own libraries in THINK C.

Compiling the program

The THINK C project manager knows when files need to be recompiled. If you edit a source file, the project manager marks it. If you edit an #include file, the project manager marks all the files that depend on it. You can ask THINK C to bring your project up to date, or you can rely on the Auto-Make facility to do it for you when you run the program.

Running the program

THINK C lets you run your program from THINK C. The project manager recompiles all the marked source files and loads any unloaded libraries. Then the THINK C linker links all your code together instantly.

THINK C launches your program as if you had double-clicked on it from the Finder. This way, you know exactly how your program will behave in actual conditions. If you're running under MultiFinder, THINK C launches your program in its own partition. Since THINK C is still running, you can look at your source files while your program is running.

Debugging the program

To help you get your program working correctly, you can use THINK C's source level debugger. The debugger lets you step through your code, set breakpoints, and examine and modify variables. You can set conditional breakpoints that stop execution only when certain conditions are true.

Building the application

Finally, when you're ready to put together the final application, THINK C's smart linker examines the object code in your project to make the final file as small as possible.

Using THINK C

The best way to learn THINK C is to follow one of the tutorials in Part Two of this manual. The rest of the chapters in this part of the manual describe the components of THINK C in detail. This summary shows you the basic steps you'll take when you write a program in THINK C.

Step	Action
Create a folder for your project	Use the Finder's **New Folder** command in the **File** menu to create a folder for your project. This folder will contain the project and all the source files your project needs.
Start THINK C	Double click on the THINK C icon from the Finder.
Create a new project	When THINK C starts, it will ask you (with a standard file dialog) to open an existing project or to create a new one. Click on the New button. A second standard file dialog will appear. Move to the folder you created for your project, name your project, and click on the Create button.
Create source files	Use the **New** command in the **File** menu to get an empty edit window. To open existing source files, use the **Open...** command in the **File** menu.
Save source files	Use the **Save** command in the **File** menu to save source files. Source files must end in `.c` for THINK C to use them in a project.

Step	Action
Add source files to the project	Use the **Compile** command in the **Source** menu to compile the active source file and add it to the project window automatically. If you don't want to compile a source file, but you still want to add it to the project window, the **Add** command in the **Source** menu will add the active source file. Use the **Add...** command to add source files you haven't opened with the THINK C editor.
Add libraries	Use the **Add...** command in the **Source** menu to add libraries to the project window. A standard file dialog will appear. Move to the folder that contains the library you want to add, and click on the Add button. The dialog box will reappear to let you add more libraries. When you're done, click on the Cancel button.
Run project	Use the **Run** command in the **Project** menu to run your project. If there are uncompiled or changed source files or libraries that need to be loaded, THINK C will ask you if you want to bring the project up to date. Click on the Yes button. If you want to use the source level debugger, choose the **Use Debugger** command before running your program to turn it on.
Build application	Use the **Build Application...** command to turn your project into a stand-alone application. A standard file dialog will prompt you for the name of your application.

The Project
7

Introduction

You can write applications, desk accessories, device drivers, and any kind of code resource in THINK C. This chapter tells you how to build different types of projects. The first section is about projects in general. It describes some of the internal components of projects, how to use resource files with projects, and the different types of projects. The second section tells you how to break up your project into segments. The remaining sections describe the four project types: applications, desk accessories, device drivers, and code resources.

What you should know

You should know how THINK C works. If you haven't done so, run through the MiniEdit example in Chapter 4. That chapter takes you step by step through building an application in THINK C and gives you the practical background you need. This chapter deals with the more technical aspects of projects.

If you want to write Macintosh applications, you should know about the resources that make up an application: menus, windows, dialogs, controls, etc. You should know how to build these kinds of resources with a resource editor, like ResEdit, or a resource compiler, like RMaker. If you don't know about these resources, look in *Inside Macintosh I*, and read the chapters that talk about the resources you want to build. *Inside Macintosh I*, Chapter 5, "The Resource Manager" talks about the resource manager in general.

If you want to write desk accessories or device drivers, you should be familiar with the mechanics of DRVR resources. These are a bit more complicated, and the information about desk accessories is mixed with information about drivers in general. See *Inside Macintosh I*, Chapter 14, "The Desk Manager" and *Inside Macintosh II*, Chapter 6, "The Device Manager."

To learn how to build other kinds of code resources (INITs, WDEFs, cdevs, etc.) see the section "Building Code Resources" later in this chapter. To learn how to call the Macintosh Toolbox routines, read the section "Calling the Macintosh Toolbox Routines" in Chapter 10.

Topics covered in this chapter:

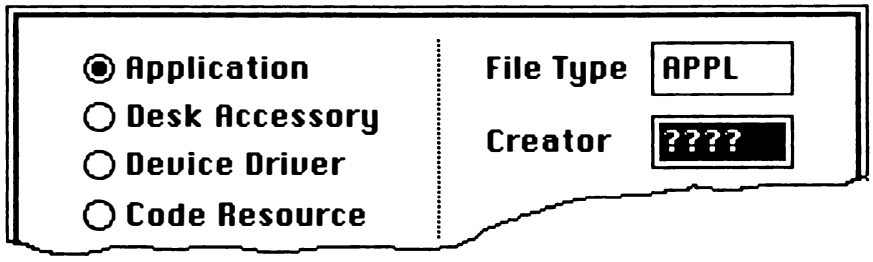
- Anatomy of a project
- Segmentation
- Building applications
- Building desk accessories and device drivers
- Building code resources

Anatomy of a Project

The project is at the heart of the THINK C development environment. It takes over the functions of several other files in traditional development environments. The project holds the object code of all your compiled source files and maintains the dependencies and connections among them. It keeps track of files that need to be recompiled or that depend on an edited #include file. And if you're using the source level debugger, the project keeps the tables that the debugger needs.

The project types

THINK C lets you build four kinds of projects: applications, desk accessories, device drivers, and code resources. To set the project type, use the **Set Project Type...** command in the **Project** menu. This command displays a dialog box that lets you set type-specific attributes for your project. For every project you can set the type and creator of the final file.



The File Type and Creator of a file let the Finder know what icon to display. Some types, like `RDEV`, `INIT`, and `cdev`, are treated specially by the Macintosh System software. To learn about File Types and Creators (also called signatures), see *Inside Macintosh III*, Chapter 1, "The Finder Interface."

There is a section in this chapter for each project type. Each section describes the type-specific settings.

The best time to set the project type is when you create a new project. You can change project types as you wish, but you'll have to recompile everything if you do so.

When you set the project type to something other than **Application**, the name of the **Build Application...** command in the **Project** menu will change accordingly. For example, if you set the project type to **Desk Accessory**, the menu will read **Build Desk Accessory...**.

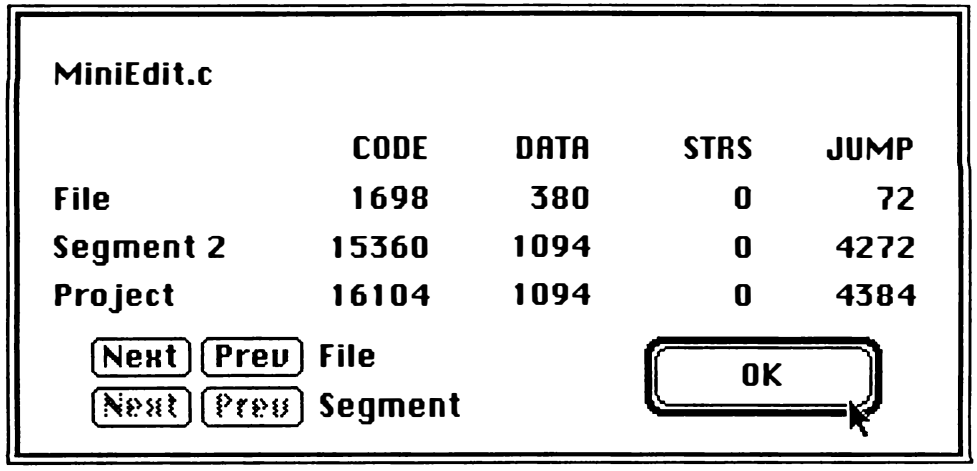
Changing the type of a project changes the way a project is built, not the way a project behaves. You can't turn an application into a desk accessory merely by changing the project type. Desk accessories are structurally different from applications. The same applies for the other types.

When you choose one of the **Build...** commands (**Build Application...**, **Build Desk Accessory...**, etc.) from the **Project** menu, THINK C creates a file and creates the CODE resources for applications, a DRVR resource for desk accessories and drivers, or the type of resource you specify for your code resource. THINK C also creates the resources it uses to manage global data and inter-segment calls.

Components of a project

Each source file or library in a project has up to four object components: CODE, DATA, STRS, and JUMP. The CODE component contains the executable code generated for the project. The DATA component contains the global and static variables. In applications, string literals and floating-point constants can be stored separately in the STRS component. Finally, the JUMP component contains the jump table, which multi-segment projects use to determine the location of a routine. Not all project types use all four components.

To examine the sizes (in bytes) of each of the components, select a source file in the project window and choose the **Get Info...** command in the **Source** menu. The display also shows the segment and project totals.



For some components there is a small amount of overhead per segment or per project.

Depending on the kind of project you build, there are different limits on the sizes of these components.

If project is a(n)...	**The limits are...**
Application	CODE: 32K per segment DATA: 32K per project JUMP: 32K per project STRS: unlimited (if you use them)
Single-segment desk accessory or device driver	CODE: 32K per project DATA: 32K per project JUMP is not used for single segment drivers
Multi-segment desk accessory or device driver	CODE: 32K per segment DATA + JUMP: 32K per project
Single-segment Code resource	CODE + DATA: 32K per project JUMP is not used for single segment code resources
Multi-segment Code resource	
Most segments:	CODE: 32K
For segment with `main()`:	CODE for segment + DATA for project + JUMP for project : 32K

If you exceed any of these limits, you'll see an error at link time.

THINK C generates an 8 byte JUMP table entry for each function that is not declared `static` as well as for each function in a non-call context (i.e. whose address is taken). The jump table built by the **Build ...** command may be smaller; the entry for each function that is only referenced within a single segment is removed.

How THINK C puts projects together

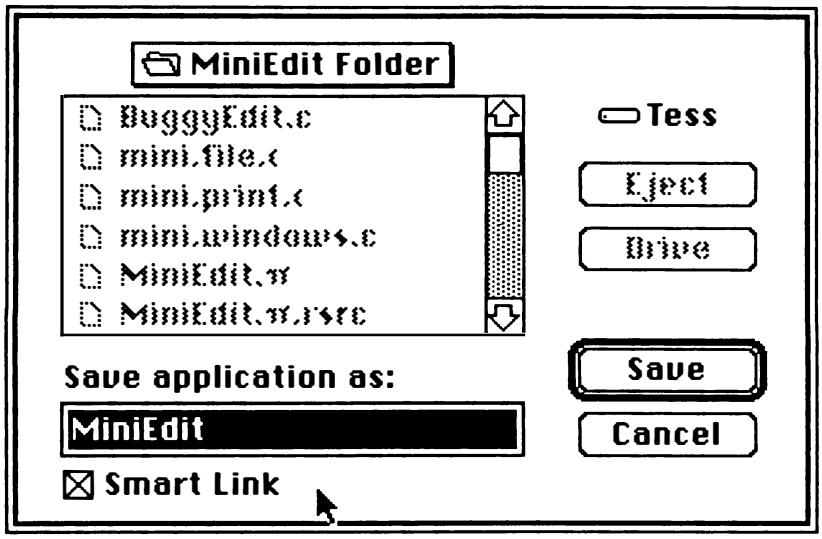
When you choose a **Build...** command from the **Project** menu, THINK C puts all the pieces of your project together as compactly as possible. THINK C uses a technique called **smart linking** to extract only the CODE elements it needs to build your application, desk accessory, device driver, or code resource.

THINK C searches the project for files or libraries whose code is never referenced and ignores them during the link. If you use a project as a library, THINK C takes only the code it needs from the project. If you use a library that you converted from a `.rel` file or one that you built with the **Build Library...** command, all of its code will be included in the final file.

Smart linking yields smaller code, but it takes several seconds longer to produce the final file. You can choose not to use smart linking; your files will be larger, but they'll come out faster.

> **Note:** For desk accessories, code resources, and applications, THINK C checks the components' size limits (described above) *after* smart-linking.

To turn smart linking off, choose your **Build...** command from the **Project** menu as you normally do, then click on the Smart Link check box to clear the option. (It's on by default.)



After THINK C finishes linking the object code, it produces the application, desk accessory, device driver, or code resource file. Finally, it copies any resources you put in the project resource file into the final file.

> **Note:** The **Build Code Resource...** dialog box has a Merge option, which builds the code resource into an existing file. If that option is checked, the project's resource file isn't copied. See "Merging code resources into files" below for more information.

Using resource files with projects

Most Macintosh applications use resources for menus, window templates, dialogs, etc. THINK C makes it easy to use resources with your applications.

Use ResEdit or RMaker to create a file that contains the resources your program needs. Save the resource file with the same name as your project plus `.rsrc`. For instance, if the name of your project is `MyProject`, name your resource file `MyProject.rsrc`. If your project

is named `SuperWizzy.project`, THINK C will look for the project's resources in `SuperWizzy.project.rsrc`.

Make sure the resource file is in the same folder as your project so THINK C can find it.

When you choose **Run** from the **Project** menu, THINK C opens the resource file automatically, so your program can access its resources.

When you choose one of the **Build...** commands from the **Project** menu (**Build Application...**, **Build Desk Accessory...**, etc.) THINK C copies the resource file into the finished application.

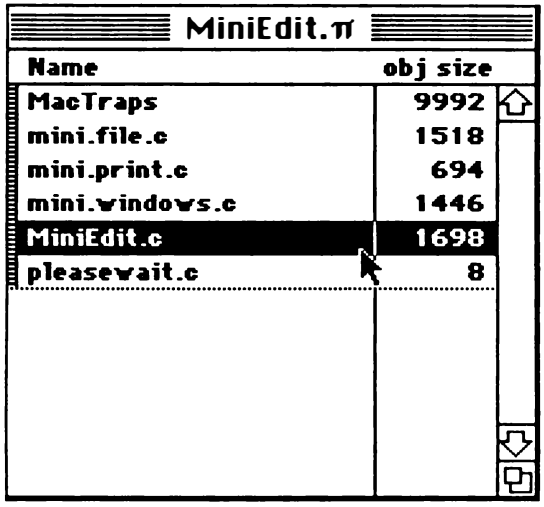
> **Note:** The **Build Code Resource...** dialog box has a Merge option, which builds the code resource into an existing file. If that option is checked, the project's resource file isn't copied. See "Merging code resources into files" below for more information.

Segmentation

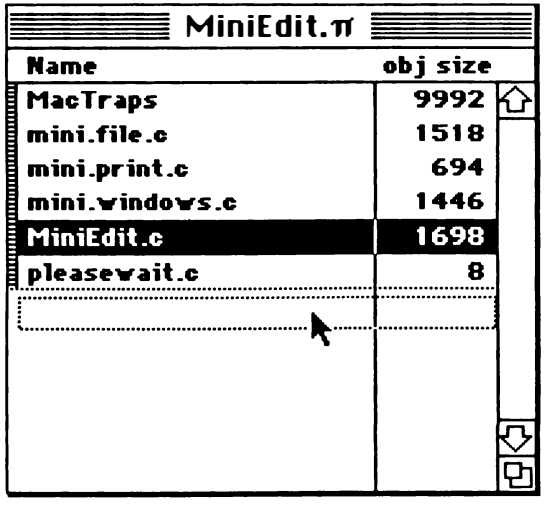
Most Macintosh programs are made up of several **segments**. Segments are units of object code that can be swapped in and out of memory as needed. The Macintosh Operating System limits segments to 32K, so if you're writing a large program, you will have to segment your code. To learn more about segments, see *Inside Macintosh II*, Chapter 2, "The Segment Loader."

THINK C lets you segment not only applications, but also desk accessories, device drivers, and code resources as well. Applications can have up to 254 segments. Desk accessories device drivers, and code resources can have up to 30 segments.

Dotted lines separate segments, and the current segment has gray hatching running along the left edge. When you add files to your project, they're added to the current segment.



To move a source file into another segment, click on its name in the project window and drag it below the dotted line.



Source files appear in alphabetical order within segments, so even if your program is small enough to fit in one or two large segments, you might want to break it down into smaller segments for cleaner organization.

If you want to rearrange whole segments at a time, hold down the Option key as you click and drag on any file name in the segment. The entire segment is placed before the segment where the drag ends. Moving segments affects only the project window display.

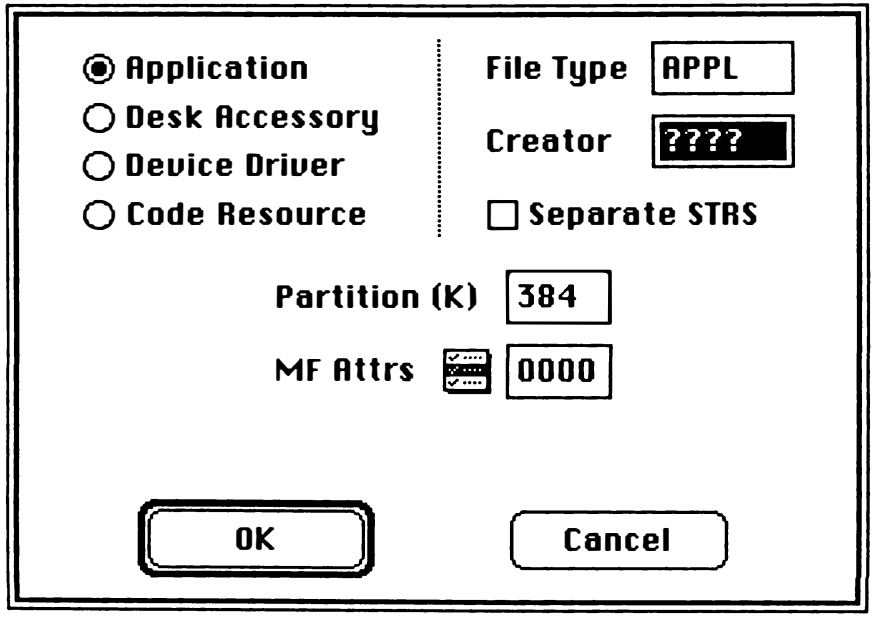
If you move all the files out of a segment, THINK C deletes the empty segment. When you use the **Add...** command, new files are placed in the current segment.

To learn more about segmentation, read *Inside Macintosh II*, Chapter 2, "The Segment Loader."

Building Applications

THINK C is set up to build applications by default. The **Set Project Type...** dialog gives you an opportunity to set some application attributes.

Choose **Set Project Type...** from the **Project** menu, and you'll see this dialog:



Setting the application file type and creator

When you're building an application, the default file type is APPL. (For applications to work with the Finder, applications must be of type APPL.) Set the Creator to whatever you want your application's signature to be. The Finder uses the file's creator to link icons with specific applications (See *Inside Macintosh III*, Chapter 1, "The Finder Interface" to learn more about applications and icons.)

Using a separate STRS component

THINK C normally places string literals and floating point constants in the DATA component of a project. When you check Separate STRS, THINK C uses a separate STRS component to store string literals and floating point constants.

When this option is off (the default), THINK C uses a 2 byte offset from register A5 to access the string literals and floating point constants. Not only is the code smaller but there's no need to do run-time address fixups. When the Separate STRS option is on, THINK C uses a 4 byte absolute address to access the string literals and floating point constants.

The only time you might want to use this option is when your DATA component is coming close to the 32K per project limit. Separating the STRS component might open up some room in the DATA component. Remember, there's no limit to the size of the STRS component.

If you build a library (or use a project as a library) with the Separate STRS option on, you can use it only in projects that have the option on as well. To ensure compatibility for both cases, leave the option off when you build a library.

Setting the partition size and MultiFinder attributes

THINK C uses the values you set in the Partition and MF Attrs (MultiFinder Attributes) fields to build the SIZE resource your application needs to run under MultiFinder.

The **partition size** determines how much memory your application gets under MultiFinder. The default partition size is 384K, which is more than enough for moderate size applications. You'll want to use a higher value for larger applications or a lower value for small applications when memory is tight.

The MF Attrs sets the flag that tells MultiFinder how compatible your application is. To learn how to make your application MultiFinder compatible, you can order the *Programmer's Guide to MultiFinder* from APDA. (See Chapter 1 for more information about APDA.)

You can type in the value of the flag (in hexadecimal) in the field, or you can set the bits from the pop-up menu next to the field. The pop-up menu looks like this:

If the MultiFinder-Aware bit is set, MultiFinder expects you to conform to the MultiFinder guidelines for shifting from the foreground to the background layers. Your application will get suspend/resume events as your application shifts from foreground to background but not activate/deactivate events.

> **Note:** If MultiFinder Aware is checked, Suspend & Resume events should be checked as well.

If the Background Null Events bit is set, your application gets regular null events when your application is in the background. Otherwise, your application gets only update events.

If the Suspend & Resume Events bit is set, your application will get these events as it shifts from the foreground to the background layers in addition to the activate/deactivate events you normally receive.

> **Note:** If you're using the THINK Class Library, be sure you check MutiFinder-Aware and Suspend & Resume Events.

Running the project

The **Run** command in the **Project** menu lets you run your application project as you work on it. When you choose the **Run** command, THINK C launches your application as if you had opened it from the Finder. If you're using MultiFinder, your application runs in its own partition.

Under MultiFinder, you can watch your application run and examine your source code at the same time. You can switch back and forth between your application and THINK C to edit your source files. When you quit your application, you'll be back in THINK C, and the auto-make facility will be ready to recompile your changed files. While your application is running, the **Run** command changes to **Resume**. Choosing **Resume** brings your application to the foreground.

> **Note:** When you're running under MultiFinder, you have to be a little more careful about stray pointers and much more conscientious about backups. A pointer to random memory may be pointing into another application's partition. A stray pointer to THINK C's data structures may damage your project, and the damage will be copied out to disk. Be careful with stray pointers. If this happens to you, delete the damaged project, and start over with your backup and bring it up to date with the Use Disk button in the **Make...** dialog.

Building Desk Accessories and Device Drivers

Desk accessories and drivers are structurally identical; they're both **drivers**. According to *Inside Macintosh*, drivers don't behave much like applications and have a different internal structure. In this section, the word *driver* by itself means either a device driver or a desk accessory.

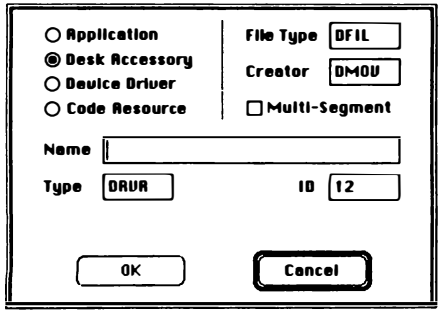
This section won't teach you how to write a desk accessory or a device driver from scratch. To learn how to write these, read *Inside Macintosh I*, Chapter 14, "The Desk Manager,"

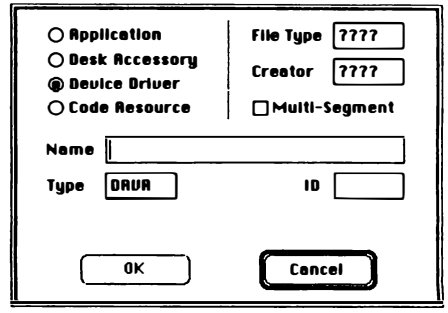
Inside Macintosh II, Chapter 6, "The Device Manager," and *Inside Macintosh V*, Chapter 23, "The Device Manager."

Setting the project type

Set the project type before you start working on a driver. If you set the project type after you've started compiling code, you'll have to recompile your source files and reload your libraries.

Choose **Set Project Type...** from the **Project** menu. When the project type dialog appears, click on either the Desk Accessory button or the Device Driver button.





The dialog boxes are the same, except for the default settings of some fields. The Desk Accessory dialog presets the File Type and Creator so the resulting file will be a Font/DA Mover file. The ID is set to 12. The Font/DA Mover renumbers it when you install it in your System. You shouldn't need to change the ID number. All that's left to do is to name the desk accessory and write the code. By convention, desk accessory names begin with a null. THINK C provides the null for you automatically.

The Device Driver dialog leaves all the options empty for you to set. Device driver names begin with a period. If you don't provide one in the Name field, THINK C automatically provides one for you.

The default settings in the project type dialogs aren't the only difference between desk accessories and device drivers. See "Setting the fields of a driver's header" below to learn about the internal differences between desk accessories and device drivers.

Both device drivers and desk accessories can have more than one segment, just like applications. Just click on the Multi-Segment check box. You can learn more about multi-segment drivers below, but read how a driver works first.

> **Note:** To use objects in a driver, you must turn on the Multi-Segment option, even if your driver has only one segment.

How drivers work

The Device Manager expects drivers to have five entry points and to be written in assembly language. When the Device Manager calls a driver written in THINK C, a short assembly-language stub (or glue routine) translates the Device Manager's request into a call to the C function `main()`. THINK C automatically places the driver glue at the beginning of your driver.

THINK C does two things to make writing a driver easier. First, it sets up a data area so that your driver can have its own global variables. Second, it figures out the proper way to return control to the Device Manager automatically. These services are discussed later in this section.

How to write main() for a driver

The driver's `main()` function takes three arguments. The function returns an `int` and is *not* declared `pascal`. A typical driver skeleton looks like this:

```
main (cntrlParam *paramBlock, DCtlPtr devCtlEnt, int n)
{
    switch (n) {
        case 0 : /*   Open    */
        case 1 : /*   Prime   */
        case 2 : /*   Control */
        case 3 : /*   Status  */
        case 4 : /*   Close   */
    }
}
```

`ParamBlock` is a pointer to an I/O parameter block. This is the value that is passed in address register `A0` to the assembly-language entry point of the driver.

`DevCtlEnt` is a pointer to the driver's device control entry. This is the value that is passed in address register `A1` to the assembly-language entry point of the driver.

`N` is a selector that specifies which entry point actually received the call. Use the value of `n` to dispatch control to the appropriate routine.

Getting the event record pointer from paramBlock

According to *Inside Macintosh I*, Chapter 14, "The Desk Manager", the `csCode` field of the `paramBlock` passed to your driver specifies what kind of action your driver should take.

When `paramBlock->csCode == accEvent`, the `csParam` field of `paramBlock` contains a pointer to an `EventRecord`. Since `csParam` is defined as an array of `int`s, this is how you cast the field to get a pointer to the event record (assume that `eventPtr` is declared `EventRecord *`):

```
eventPtr = (*(EventRecord **) paramBlock->csParam)
```

Global data in drivers

You can declare global and static variables in drivers. The THINK C driver glue allocates the space for the globals in the heap before it calls `main()` to implement the `Open` entry. The glue releases the memory when the driver returns from a `Close` call. (There is a way to keep the global data area allocated after a `Close` ; see "Returning from a driver" below.)

> **Note:** In drivers, string literals and floating point constants are stored the same way as global variables.

Macintosh applications use register `A5` to access their globals. Since drivers co-exist with running applications, they can't use register `A5` to access their globals. Instead, drivers use register `A4`.

The THINK C driver glue stores a handle to the dynamically allocated data area in the `dCtlStorage` field of the driver's device control entry. This handle is dereferenced into address register `A4` and locked before each call to `main()`. Your `Open` routine must check whether the data area was allocated successfully. If it was not, the `dCtlStorage` field will be 0, and your driver should display some error message (without using any globals!) and close itself.

The data area remains locked between calls to your driver. If you like, you can unlock it yourself before returning. If you unlock the data area, though, make sure that you don't rely on the address of any data item staying the same between calls. Also, make sure that the data area doesn't contain any objects, such as windows, that the Toolbox assumes will not move.

Using driver globals in callback and trap intercept routines

The THINK C driver glue sets up register `A4` for you whenever it's called from `main()`. If your driver defines callback routines, trap intercept routines, or other functions that might be called when the value of `A4` is in doubt, you have to save `A4` where your routines can find it.

The #include file `SetUpA4.h` defines a set of macros that take care of saving, setting, and restoring `A4` for you.

Suppose your driver calls `ModalDialog()` with a `filterProc`. Since you're not sure if the value of `A4` will be correct when `ModalDialog()` calls your `filterProc`, you need to save it. Your `filterProc` needs to set `A4` to the saved value and then restore it before it exits. Your call to `ModalDialog()` would look like this:

```
engage_in_dialog()
{
    extern pascal Boolean myFilter();
    int item;
    ...
    RememberA4();
    ModalDialog(myFilter, &item);
    ...
}
```

Your `filterProc` would look like this:

```
pascal Boolean myFilter(DialogPtr dp, EventRecord eventp, int *itemp)
{
    Boolean result;

    SetUpA4();
    ...
    RestoreA4();
    return(result);
}
```

The calls to `RememberA4()` and `SetUpA4()` must appear in the same source file.

Use the same technique for trap intercept routines that need access to a driver's globals. Of course, if your callback or trap intercept routine doesn't use driver globals, you don't need to set up and restore `A4`.

Using libraries in drivers

You can use libraries in drivers as long as the libraries don't reference global variables accessed through register `A5`.

The `MacTraps` library doesn't reference any globals, so you can use it in your drivers. `MacTraps` does define the QuickDraw globals, though. If you access these globals from a driver, you'll find that they aren't the real QuickDraw globals but simply the driver's own variables that QuickDraw knows nothing about.

You can access the real QuickDraw globals from a driver by observing that `0(A5)` holds the address of the last of the QuickDraw globals, `thePort`. The remaining QuickDraw globals

are at descending addresses from `thePort`; refer to *Inside Macintosh I*, Chapter 6, "QuickDraw" for more information. The value of `A5` is stored in the low-memory global `CurrentA5`. You might want to use the inline assembler to get to the QuickDraw globals. See Chapter 12 for details.

Other libraries supplied with THINK C reference their globals from register `A5`, so you must modify them. To make a library use register `A4` for its globals, first make a copy of the library. Then change the project type of the library to anything but Application, and recompile and rebuild the library. See Chapter 13 to learn how to build libraries.

Setting the fields of a driver's header

A driver begins with a header containing several flags and other data items (some of which apply only to desk accessories). When the driver is opened, the Device Manager copies these fields to the device control entry before the `Open` entry point is called. After that, the device header is not used. The fields in the driver are used only to initialize the fields in the device control entry.

THINK C presets these fields to reasonable default values. If your device driver or desk accessory requires different settings for these fields, modify them on the fly in the device control entry.

These are the default settings for desk accessories:

Field	**Value**

`dCtlFlags`	`dReadEnable 0`
	`dWritEnable 0`
	`dCtlEnable 1`
	`dStatEnable 0`
	`dNeedGoodbye 0`
	`dNeedTime 0`
	`dNeedLock 0`

`dCtlDelay`	`N/A because dNeedTime = 0`
`dCtlEMask`	`0x016A (mouseDown, keyDown, autoKey, update, activate)`
`dCtlMenu`	`0`

These are the default settings for device drivers:

Field	Value
dCtlFlags	dReadEnable 1
	dWritEnable 1
	dCtlEnable 1
	dStatEnable 1
	dNeedGoodbye 0
	dNeedTime 0
	dNeedLock 1
dCtlDelay	N/A because dNeedTime = 0
dCtlEMask	N/A (desk accessories only)
dCtlMenu	N/A (desk accessories only)

Suppose you want a desk accessory to remain locked between calls and to be called once per second (every 60 ticks). Just include this code in the driver's Open routine. (devCtlEnt is the second argument to main(), the pointer to the device control entry.)

```
devCtlEnt->dCtlFlags |= dNeedLock|dNeedTime;
devCtlEnt->dCtlDelay = 60;
```

Opening an open driver

The Open entry point of a driver (main()'s third argument == 0) may be called even if the driver is already open. This happens, for example, when the user selects the name of a desk accessory that's already on the screen. The driver should check to see if it is already open to avoid repeating its initialization sequence.

You can set the fields of the device control entry directly as shown above, but the Device Manager copies the dCtlFlags, dCtlMenu, dCtlDelay, and dCtlEMask fields of the driver's header to the corresponding fields of the device control entry *every time* the Open routine is called, even if the driver is already open. So you need to set these fields to their proper values each time. Your Open routine might look something like this:

```
doOpen()
{
    devCtlEnt->dCtlFlags|= dNeedLock;/*  or whatever  */

    if (already_open) /* already_open is a driver global */
        return;
    already_open = 1;
    /*  one-time initialization  */
}
```

How to return from a driver

If your `Open` routine was successful, return 0. If the `Open` routine fails, return a negative result and the driver will not be opened.

Return 0 from `Close` if it was successful. If your `Close` routine returns `closeErr` (-24) the driver won't be closed. If you return a 1, the THINK C driver glue will preserve the `dCtlStorage` field of the device control entry. This way you can keep your driver globals around until your driver is reopened. (The driver glue will make it seem as though your `Close` routine returned 0, meaning the `Close` was successful.).

> **Note:** Returning negative values from `Open` and `Close` to prevent opening or closing works only on 128K and later ROMs.

Return 1 from asynchronous calls to `Prime`, `Control`, and `Status` routines if the request could not be completed right away. This result code will be stored in the `ioResult` field of the I/O parameter block, but 0 (no error) will be returned to the Device Manager.

The jIODone problem

THINK C always returns from a driver correctly. In other development systems, it's not so easy. Read this section if you want to learn about this problem. Since you don't have to worry about it, you might want to skip this section.

One of the trickiest aspects of returning from a driver is deciding whether to return directly to the Device Manager (via an `RTS` instruction) or whether to jump to `jIODone`. This is a complex issue, and many existing desk accessories do it wrong (though, fortuitously, they manage to work anyway).

Associated with each driver is an I/O queue, which is a list of I/O parameter blocks waiting for service from the driver. Calls made to a driver fall into one of two categories: *queued*, meaning that the I/O parameter block passed as an argument to the call is in the driver's queue; and *immediate*, meaning that it is not. In the immediate case, the queue may even be (and in fact usually is) empty.

All `Open` and `Close` calls are immediate. All `Control` calls made to desk accessories are immediate, except for the "goodbye kiss" (`csCode=-1`) issued to desk accessories that have requested to be notified when the current application exits out from under them. Other calls may be queued or immediate.

The rules for returning from a driver are: The driver should return directly to the Device Manager from all immediate calls. It should also return directly to the Device Manager from queued calls requesting asynchronous I/O that could not be completed right away. Finally, it should jump to `jIODone` from queued calls if the driver completed the request (or if there was an error).

It is incorrect to violate these rules; in particular, it is incorrect to jump to `jIODone` to return from an immediate call. `jIODone` will attempt to examine the driver's I/O queue, and since the queue is usually empty it will end up examining low-memory locations beginning at `0x0000`. Apparently, these locations somehow look enough like an I/O parameter block to satisfy the Device Manager, but this is clearly an unsafe situation.

Just to make things difficult, when returning from `Prime`, `Control`, and `Status` calls, it is `jIODone` that unlocks the driver's code and its device control entry so they won't form islands in the heap between calls to the driver (unless, of course, the driver has requested that they remain locked). So the author of a desk accessory, for instance, has to make a difficult decision — to return directly to the Device Manager, leaving the driver's code and its device control entry locked and potentially interfering with the host application; or to violate the rules and jump to `jIODone`. Most desk accessories seem to take the latter route.

THINK C avoids this dilemma. When a driver written in THINK C returns from `main()`, the decision whether to call `jIODone` is made automatically (and correctly). For `Prime`, `Control`, and `Status` calls, if the decision is made to return directly to the Device Manager, and the driver has not requested that its code and device control entry remain locked, they are unlocked.

Multi-Segment drivers

If the Multi-Segment option is on, drivers can contain multiple segments. As with applications, segments are loaded automatically as they are called. In addition, all loaded segments are unloaded automatically upon return to the Device Manager after each call, *unless* the `dNeedLock` bit is set in the driver's device control entry.

You can unload driver segments manually with this function:

```
void UnloadA4Seg(ProcPtr);
```

This function works just like `UnloadSeg()` does in applications.

> **Note:** Do not use `UnloadSeg()` instead of `UnloadA4Seg()` by mistake!

Read the Segmentation section above to learn how to break up a project into different segments.

Building Code Resources

You can use THINK C to write pure code resources. Code resources don't have the complex structure of drivers; they simply contain code to be called at the entry point, `main()`.

You might want to write code resources for several reasons. You might want to write a window definition function that you can use in several other programs, or you might want to

write an `INIT` to run at startup. You may define your own code resource types to make a function you've written in THINK C available to a program written in another language. The "client" program simply loads the resource and calls it at its beginning. It's up to you whether you use C or Pascal calling conventions. (For more information about calling conventions, see Chapter 12.)

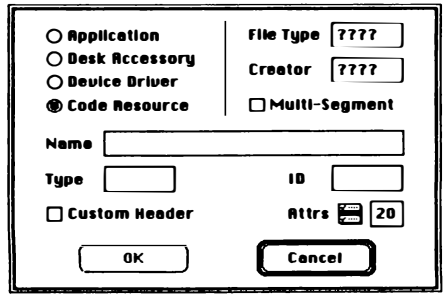
This section tells you how to build code resources in THINK C. The specific formats and calling sequences for code resources are given in the various volumes and chapters of *Inside Macintosh*. This list will help you get started.

To learn how to build a...	**Read *Inside Macintosh*...**
`ADBS` resource	Volume V, Chapter 20, "The Apple Desktop Bus"
`CDEF` resource	Volume I, Chapter 10, "The Control Manager"
`cdev` resource	Volume V, Chapter 18, "The Control Panel"
`FKEY` resource	Technical Note 3 (also see below)
`INIT` resource	Volume IV, Chapter 29, "The System Resource File" Volume V, Chapter 19, "The Start Manager"
`LDEF` resource	Volume IV, Chapter 30, "The List Manager Package"
`MBDF` resource	Volume V, Chapter 13, "The Menu Manager"
`MDEF` resource	Volume I, Chapter 11, "The Menu Manager" Volume V, Chapter 13, "The Menu Manager"
`WDEF` resource	Volume I, Chapter 9, "The Window Manager"
`XCMD` resource	Appendix A, *HyperCard Script Language Guide*
`XFCN` resource	Appendix A, *HyperCard Script Language Guide*

Setting the project type

Set the project type before you start working on a code resource. If you set the project type after you've started compiling code, you'll have to recompile your source files and reload your libraries.

Choose **Set Project Type...** from the **Project** menu. When the project type dialog appears, click on the Code Resource check box.



Fill in the Type and ID of the code resource you're building. If you like, you can give your code resource a name. Use the Attrs (attributes) pop-up menu to set the resource attributes for your code resource. If you prefer, you can enter a hex value in the Attrs field. To learn about resource attributes, see *Inside Macintosh I*, Chapter 5, "The Resource Manager."

Code resources can have more than one segment. If your code resource will have multiple segments, select the Multi-Segment option. Read the section "Multi-segment code resources" for more information.

> **Note:** To use objects in a code resource, you must turn on the Multi-Segment option, even if your code resource has only one segment.

How to write main() for a code resource

The way you write your `main()` routine depends on the kind of resource you're writing. The main point to remember is that if your code resource is going to be called from a Pascal program or as a callback routine, it must be declared `pascal`.

An `FKEY` resource, for example, is called by the Event Manager. It doesn't have any arguments. You would define `main()` like this:

```
main()
{
    ...
}
```

A `WDEF` resource is a custom window definition. The Window Manager calls `main()` with several arguments and expects the window definition function to return a `long`. This is how you would write `main()` for a `WDEF`:

```
pascal long main (int varCode, WindowPtr theWindow,
                  int message, long param)
{
    ...
}
```

Global data in code resources

Code resources, like applications and drivers, can have global and static variables. When you build a single-segment code resource, the DATA component is appended to the CODE component, so CODE and DATA together must be less than 32K. When you build a multi-segment code resource, the DATA and JUMP components from all the segments are appended to the CODE component of the segment that contains `main()`, so all the DATA and JUMP information and the CODE component for the main segment must be less than 32K.

Code resource globals are addressed as offset from A4. Unlike drivers, however, A4 isn't set up automatically for you when your main() routine is called. You have to do this yourself.

> **Note:** In multi-segment code resources, THINK C treats string literals and floating point constants the same way as globals. If you're using literals and constants but not globals, you still need to set up A4. In single-segment code resources, string literals and floating point constants are *not* treated the same way as globals. For simple code resources, you do not need to set up A4. This is different from the previous version of THINK C in which all code resources treated string and floating point literals in the same way as globals.

When main() is called, A0 points to your code resource. This is the same value that your code resource expects A4 to have to find your globals.

The #include file SetUpA4.h contains a set of macros that help you set up the A4 register. Immediately after you enter main(), call RememberA0(). This macro saves the value of A0 where another macro, SetUpA4(), can find it. You must call RestoreA4() before you return from main(). For example:

```
main()
{
    RememberA0(); /* To access resource globals */
    SetUpA4();
    ...

    RestoreA4();
}
```

> **Note:** This technique works only when you use the default code resource header. If you use a custom header, you'll have to set up A4 another way. See "Code resource headers" below to learn how to do this.

Using libraries in code resources

You can use libraries in code resources as long as the libraries don't reference global variables accessed through register A5.

The MacTraps library does not access any globals, so you can use it in your code resources. MacTraps does define the QuickDraw globals, though. If you access these globals from a code resource, you'll find that they aren't the real QuickDraw globals but simply the resource's own variables that QuickDraw knows nothing about.

You can access the real QuickDraw globals from a code resource by observing that 0(A5) holds the address of the last of the QuickDraw globals, thePort. The remaining

QuickDraw globals are at descending addresses from `thePort`; refer to *Inside Macintosh I*, Chapter 6, "QuickDraw" for more information. The value of `A5` is stored in the low-memory global `CurrentA5`. You might want to use the inline assembler to get to the QuickDraw globals. See Chapter 12 for details.

The other libraries supplied with THINK C reference their globals from register `A5`, so you must modify them. To make a library that uses register `A4` for its globals, first make a copy of the library. Then change the project type of the library to anything but Application, and recompile and rebuild the library. To learn more about libraries, see Chapter 13.

Locking code resources

The Macintosh Toolbox takes care of locking and unlocking the standard code resources like `WDEF`s. When you write your own code resources, you can either let the caller take responsibility for locking and unlocking them, or you can have the code resource do it itself.

When `main()` is entered, register `A0` contains a pointer to your code resource. If you need to lock it, you would write main like this:

```
#include <SetUpA4.h>

main()
{
    Handle h;

    RememberA0(); /* To access resource globals */
    SetUpA4();

    asm {
        _RecoverHandle      /* a0 already points to resource */
        move.l  a0, h
    }
    HLock(h);

    ...

    HUnlock(h);
    RestoreA4();
}
```

> **Note:** This technique works only when you use the standard code resource header. If you use a custom header, you'll have to get the address of your code resource another way. See "Code resource headers" below.

If your code resource can be called reentrantly, it should not unconditionally be unlocked each time it returns. Instead, it should be restored to the same state of locked-ness it had on entry.

Code resource headers

If the Custom Header option is off, THINK C creates a code resource with a standard header:

Offset	Contents
0	`BRA.S .+0x10 (branch to header code)`
2	`0x0000 (unused)`
4	`'TYPE' (resource type)`
8	`0x000A (resource ID)`
10 (0xA)	`0x0000 (unused)`
12 (0xC)	`0x0000 (unused)`
14 (0xE)	`0x0000 (unused)`

The standard header code puts the address of your code resource in register `A0` and then branches to your `main()` routine, but the file containing `main()` is *not* guaranteed to be the first file in the code resource. You can do anything you like with the unused words.

> **Note:** Older versions of THINK C put the address of your code resource in the low memory global `ToolScratch`. The standard code resource header does not do this. Use inline assembly if you need to reference `A0` directly.

If the Custom Header option is checked, THINK C does not generate the standard header. Instead, your code resource starts with the first function in the file where `main()` is defined. The routine `main()` doesn't have to be the first function in the file, so your resource can begin any way you like. (In fact, `main()` may never be called.) This option is useful for certain types of code resources that must begin with a table of some kind, rather than with code.

When you use a custom resource header, `A0` does not contain the address of your code resource. This means that you can't use the `RememberA0()` macro to set up register `A4` to use your code resource globals until you set up `A0` to point to your code resource.

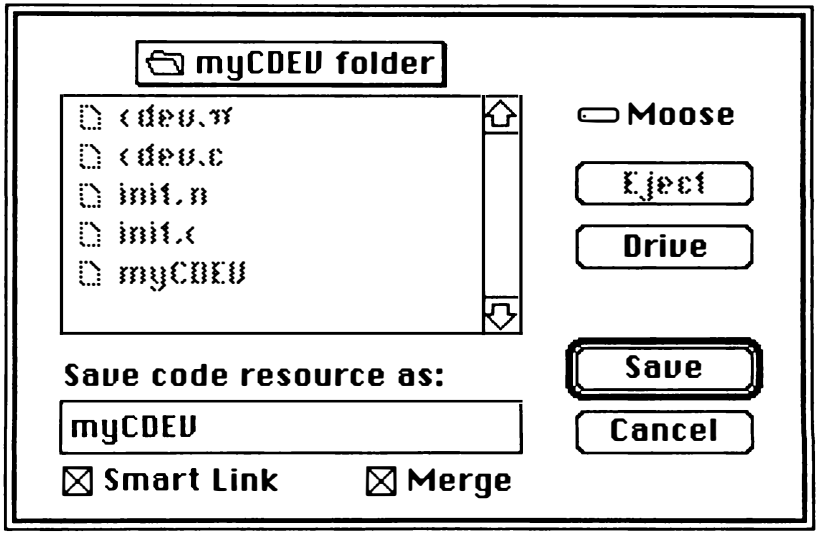
The following code shows one way to set up your code resource globals when you use a custom resource header.

> **Note:** `SetUpA4.h` generates code, so if you #include it at the top of your file, the internal function defined in `SetUpA4.h` will be your header. This is not what you want.

```
/* #includes, globals, declarations */

extern main();  /* declare it so we can JMP to it */

header()  /* First function in file */
{
    asm {
    DC.L        0   /* header information */
    ...
    /* end of header */
    LEA header, a0 /* put address of code resource in a0 */
    JMP main
    }
}

/* SetUpA4.h generates code, so don't put it at the    */
/* top of the file with the other #includes.         */
#include <SetUpA4.h>

main()
{
    RememberA0();
    SetUpA4();

    ...

    RestoreA4();
}
```

Merging code resources into files

Unlike other project types, you can build code resources into already existing files. In the **Build Code Resource...** dialog, check the Merge option, and type the name of the file you want your code resource placed into.



You might want to do this when you're building a library of XCMDs or building a combination cdev and INIT. However, when you check the Merge option, the project's resource file will not be copied into the file, so you should make sure that any resources needed by your code resource are already present in the file.

Multi-segment code resources

If the Multi-Segment option is checked in the **Set Project Type...** dialog, code resources can contain multiple segments. As with the other project types, segments are loaded automatically as they are called. However, unlike the other project types, you must unload the segments yourself when the code resource exits. You can unload individual segments with this function:

```
void UnloadA4Seg(ProcPtr);
```

This function works just like `UnloadSeg()` does in applications. To unload all segments at once, call `UnloadA4Seg()` with a null pointer; that is, `UnloadA4Seg(0L)`.

> **Note:** Do not use `UnloadSeg()` instead of `UnloadA4Seg()` by mistake!

Read the Segmentation section above to learn how to break up a project into different segments.

In code resources, the segment that contains the `main()` routine is special. The DATA and JUMP components for all the segments is appended to the CODE component of the main segment. So all the DATA and JUMP information and the CODE component of the main segment must be under 32K.

The Editor
8

Introduction

This chapter describes the features of the built-in THINK C editor. The editor uses standard Macintosh editing techniques so you're familiar with its basic operation. Although you can use it to edit any text file, the editor has some features that make editing your source and #include files easier.

Topics covered in this chapter:

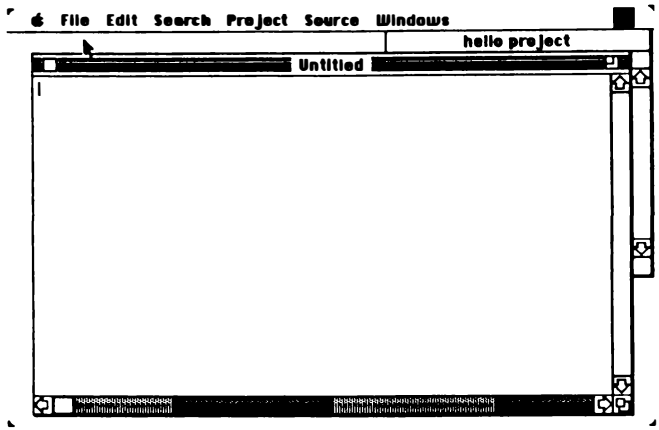
- Creating and opening files
- Editing text
- Printing files
- Closing and saving files
- Searching and replacing text

Creating and Opening Files

To create or open a file, you must have a project window open. You can open as many files as the memory in your Macintosh will allow, and each file appears in its own edit window. Although you usually create and open source or header files, you can also use the THINK C editor to open any text file.

Creating a new file

To create a new file, select **New** from the **File** menu. An untitled edit window will appear, ready for you to start typing into it.



Opening a text file

The **Open...** command in the **File** menu opens any text file. A standard file dialog box displays the names of all text files in the current folder, even if they weren't created with THINK C. The file you open appears in its edit window.

Opening a source file

To open a source file that's already in your project window, just double click on its name in the project window. If the file is already open, double clicking on its name brings the file's edit window to the front.

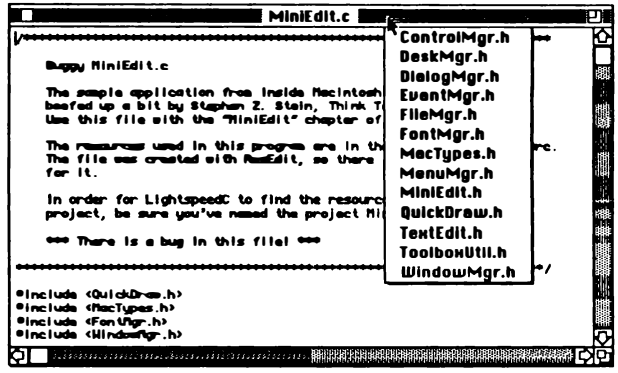
You can open a file by typing the first portion of its name and then pressing Return or Enter when the project window is the active window. Since the files in the project window are in alphabetical order, the selected file is the first file in the project which matches the characters typed so far. If what you type doesn't match any name in the project window, any selected file is deselected. When what you type matches more than one file name, you can use the Tab key to cycle among all the names that match. The up- and down-arrow cursor keys also change the selection in the project window.

> **Note:** In the project window, you can use Backspace to mean up-arrow, and Shift-Backspace to mean down-arrow.

Opening #include files

Sometimes when you're working on a source file, you want to look at the #include files associated with it. Naturally, you can use the **Open...** command in the **File** menu to open #include files, but THINK C gives you a way to open these files quickly.

Hold down the Option or Command key as you click in the title bar of a source file's edit window to get a pop up menu of all the files included in the source file. Choose the #include file you want to look at and it will appear in its own edit window. If the file is already open, its window will be brought forward.



Holding down the Option or Command key as you click in the title bar of the project window brings up a pop up menu containing the names of all the #include files used in the project.

THINK C builds the pop up menus only for compiled source files. Any #include files that are part of a precompiled header don't appear in the pop up menus. (To learn more about precompiled headers see Chapter 10.)

If you want to open an #include file that you've added to the source file since the last compilation, or if you want to open an #include file for a file you haven't compiled yet, use the **Open Selection** command described below.

Opening the current selection

The **Open Selection** command in the **File** menu lets you open an `#include` file by selecting its name in the current source file (double clicking on the name works here). You don't have to select the `.h` extension. The selection will automatically be extended to include it as long as you've selected the first part of the file name.

Open Selection can deal with path names. If the character following the selection is `.` (period) or `:` (colon), the selection is extended to the end of the next word following, and this process is repeated. A partial path name is searched for as though it appeared in an #include "..." statement in the file being edited. (**Open Selection** is not smart enough to look for angle brackets.) If the editing window is untitled, only the project and THINK C trees are searched. (To learn about the project and THINK C trees, see Chapter 9.)

Editing a File

The editor uses all the standard Macintosh editing techniques as well as some designed for editing C programs. Double clicking on a word will select the entire word. **Cut**, **Copy**, **Paste**, and **Undo** all work the way they do in other Macintosh applications.

Typing text

The THINK C text editor does not have the word wrap feature you might be used to in other editors. If you type past the right edge of the window, use the horizontal scroll bar at the bottom of the window to see past the right edge.

Undoing changes to a file

If you make an unintentional change to your file, use the **Undo** command in the **Edit** menu. Undo remembers only the last thing you did. You can also use Undo to Redo what you undid. The wording of the Undo command always reflects what it will undo. For instance, if you cut a range of lines, the Undo command will read Undo Cut. If you select Undo, the command will now say Redo Cut. You can undo the most recent replace (see Searching and Replacing in the next section), but you can't undo a Replace All command.

If you've made numerous changes to your file, and you want to undo all of them, or if you want to undo a Replace All, use the **Revert** command in the **File** menu. Revert discards all of the changes you've made since you last saved (or opened) the file.

Scrolling to the insertion point

THINK C includes a feature that allows you to examine other areas of the text, then instantly jump back to where you were. Pressing the Enter key while you are anywhere in the text will reposition the text to show the current insertion point or the start of the current selection. Since scrolling in any direction does not affect the insertion point or the current selection, this will take you back to your previous position in the file.

If the start of the selection is already visible, pressing the Enter key makes the end of the selection visible, so you can toggle between the two ends of the current selection. This is particularly useful after using the **Balance** command.

Using the arrow keys

The arrow keys on the Macintosh Plus, Macintosh SE, and Macintosh II keyboards move the insertion point up, down, left, and right. At the ends of lines, the left and right arrow keys wrap around. In other words, pressing the left arrow key at the beginning of a line moves the insertion point to the end of the previous line, and pressing the right arrow key at the end of a line moves it to the beginning of the next line. Pressing the Option key with any arrow key moves the insertion point as far up, down, left, or right as possible. In other words, Option-up moves the insertion point to the beginning of the file; Option-down moves it to the end of the file; Option-left moves it to the beginning of the line; and Option-

right moves it to the end of the line. Shift-arrows and Shift-Option-arrows extend the current selection.

> **Note:** Because of the Macintosh keyboard hardware design, it's not possible to distinguish between Shift-arrows and the +, *, /, and = keys on the numeric keypad of the Macintosh Plus keyboard. For example, when you type a + on the keypad, it will be treated as a Shift-left arrow. The shifted versions of these keys are the same as Shift-arrow combinations. On the Mac SE and Mac II, the +, *, /, and = keys work correctly.

Selecting lines

To select a line, triple click anywhere on the line. To select a range of lines, drag the mouse after you triple click.

Indenting

If you indent a line with leading tabs or spaces, the editor will indent the following lines by the same amount of space.

To keep a line from auto-indenting, hold down the option key when you press Return. The editor uses tabs and spaces to indent the line, so you can just backspace over them if you want to change the indentation.

Shifting blocks right and left

To change the indentation level for a range of lines use the **Shift Left** and **Shift Right** commands from the **Edit** menu. **Shift Left** deletes the leading tab from each selected line. **Shift Right** inserts a tab at the beginning of each selected line. Both **Shift Left** and **Shift Right** extend the current selection to include entire lines.

> **Note:** Do not type anything while doing **Shift Left** or **Shift Right** on a selected region. This will, of course, replace the selected text with what you typed.

Balancing parentheses, brackets, and braces

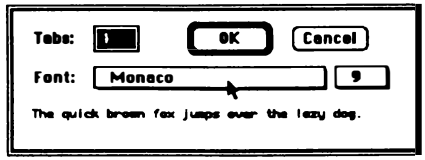
The **Balance** command in the **Edit** menu extends the current selection in both directions until it encloses the smallest surrounding balanced text enclosed in parentheses `()`, brackets `[]`, or braces `{}`. Successive invocations select larger sequences of text.

> **Note:** The **Balance** command just scans for matching characters without taking into consideration whether the match is inside a string or a comment.

Try this: Start at the beginning of a file and search for the first left brace `{`. Then use **Balance** and **Find Again** commands repeatedly until you get to the end of the file. This is a quick way to check whether all your function definitions are properly balanced.

Changing font and tab settings

When you open a new edit window with the **New** command, the font is preset to Monaco-9, and the tab stops are set to every 4 spaces. You can change these settings with the **Set Tabs & Font...** command in the **Edit** menu. When you choose this command, you'll see this dialog:



Type a number to set the number of spaces per tab. To change the font, click on the font name. You'll see a pop up menu with the names of the fonts in your System file. Click on the font size to see a pop up menu of the sizes for the font.

If you are using a proportionally spaced font like New York or Geneva, the THINK C editor uses the width of the non-breaking space (Option-Space) to figure out the width of a tab.

When you change the font or tab settings, the editor adds `EFNT` and `ETAB` resources to your text files to record the new settings. Other text editors use these resources as well.

> **Note:** To change the default font and tab settings, use ResEdit to modify the `CNFG 0` resource in the THINK C application. The second, third, and fourth words of this resource specify the font number, font size, and tab width, respectively, used for newly created Untitled windows.

Printing Files

Use the **Print...** command in the **File** menu to print the file in the frontmost edit window. You'll see the standard print dialog for either the ImageWriter or LaserWriter.

The **File** menu also contains a standard **Page Setup...** command that lets you set the page size and other options before you print.

Closing and Saving Files

It's a good idea to save your work every fifteen minutes or so just in case something horrible happens. Power failures usually come at the most inopportune times, and strange machine crashes do occur.

Closing a File

To close a file, click on its edit window's close box. If you've made changes, and you haven't saved the file, the editor will ask you if you want to save it before closing. You can also use the **Close** command in the **Edit** menu to close a file.

Saving a File

To save a file without closing it, use the **Save** command from the **File** menu. If you've never saved the file before (that is, if its edit window is untitled), you'll get a standard file dialog asking you to name the file.

Saving a File With a Different Name

The THINK C editor gives you two ways of saving a file under a different name. The **Save As...** command asks you to name a new file and then saves the contents of the edit window under that name. If the file is part of your project (it's in your project window), its name will change there, too.

The **Save a Copy As...** is similar to the **Save As...** command except that it doesn't change the name of the file. This command files the contents of the current edit window under a new name, but lets you continue editing the original file.

Saving and closing all open files

To save all the open files, use the **Save All** command in the **Windows** menu. This command is the same as using the Save command on each window.

The **Close All** command closes all the open files. If a file hasn't been saved, the editor will ask you if you want to save it.

Saving Files Automatically

THINK C will save files for you automatically when you close them if you uncheck the Confirm Saves check box in the Preferences section of the **Options...** command. See Chapter 55 for more information about the **Options...** command.

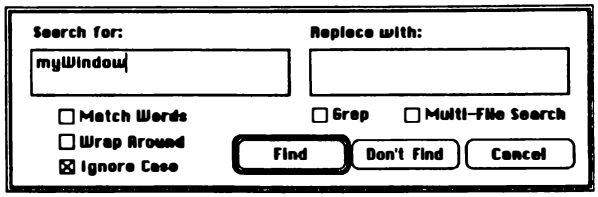
Searching and Replacing

The THINK C editor offers a wide range of search and replace capabilities. You can:

- find a string in a file
- replace one string with another
- find a string in any file in your project
- find the definition of a symbol
- find strings that match a pattern (grep)

Finding a String

Use the **Find...** command in the **Search** menu when you want to find a string. You'll see this dialog box:



Type the string you're looking for in the Search for: field and click the Find button. If the string is in the file you're editing, it will be highlighted. If it's not, the editor just beeps.

Since the string you've found is highlighted, you can replace it just by typing in a replacement string.

To find the next instance of the string, use the **Find Again** command in the **Edit** menu.

Search options

The three check boxes on the left side of the Find dialog let you specify how the editor looks for your string. You can set the defaults for these options with the **Options...** command in the **Edit** menu.

If you check the Match Words option, the editor will match only whole words. This option is useful when you're looking for one-letter variable names, for instance.

The editor usually searches from the insertion point to the end of the file. If you check the Wrap Around option, it will search the entire file for your string. The search begins from the insertion point (or the end of the selection). If your string hasn't been found by the time the editor gets to the end of the file, it searches from the beginning to the insertion point.

When the Ignore Case option is checked, the editor will match the search string regardless of case. If this option is off, the case of the strings must match exactly.

Replacing a String

If you want to replace some but not all instances of the search string, enter a replacement string in the Replace with field of the Find dialog box. Then, when the editor finds the first occurrence, you can use the **Find Again** command to go on to the next instance, **Replace**, to replace it with the replacement string, or **Replace and Find Again** to replace the current instance and then immediately go on to the next one.

Replace All replaces every instance of the search string in the file with the replacement string. If you don't type in a replacement string, it will delete every instance of the search string (that is, it will replace it with nothing).

Setting things up for searching later

In addition to the Find button ("Go ahead with the search") and the Cancel button ("Pretend I never invoked this command"), there is a Don't Find button. Clicking on this button sets up the search and replace strings and the option settings without doing the search.

You might want to use this button when you realize that the insertion point is not where it should be to start the search. Click on the Don't Find button, move the insertion point to the proper place, then use the **Find Again** command to find your string.

> **Note:** The **Find Again** command looks for the string you've entered in the search string.

Finding Non-printing Characters

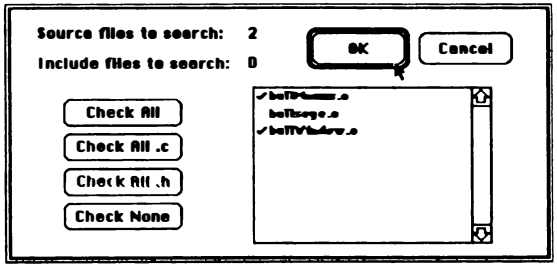
To look for tab and return characters, hold down the Command key as you type them into the Search for and Replace with fields. To insert other non-printing characters, use the **Copy** command to copy them into the Clipboard, then **Paste** them into the Search for and Replace with fields.

A return signifies the end or beginning of a line in a string search.

Searching Through Multiple Files

The Multi-File Search option lets you look for a string in more than one file. This feature is useful when you're tracking down undefined or multiply defined symbols, or if you change the number of parameters to a function, and you need to fix up all the references to it.

To look for a string in more than one file, check the Multi-File Search check box in the **Find...** dialog box. When you check this box, another dialog box displays all the text files associated with the project.



Scroll through the list and click on individual files to select them. A small check mark appears next to the file name.You can use the buttons in the dialog box to Check All, Check

None, Check All .c, or Check All .h files. (If a file is already selected, clicking on its name will remove the check mark.)

> **Note:** Typing Command-A in the **Find...** dialog box is the same as clicking on the Check All .c button.

When you've checked the files you want to search, click OK to return to the Find dialog box., then click Find to start the search.

THINK C looks for the search string through each of the checked files, starting with the first one checked. When it finds a file that contains the search string, THINK C opens the file and selects the search string. At this point, you can edit the file, or, if you want to search further in the current file, you can use the **Find...**, **Find Again**, **Replace**, and **Replace All** commands to work within the current file. When you're ready to go on to the next file, use the **Find in Next File** command.

> **Note:** Multi-file search finds the first instance of the search string in each file. To find subsequent instances of the search string in the file, use the **Find Again** command. Once you issue a **Find in Next File** command, THINK C will look in the next file you checked, even if there are additional instances of the search string in the current file.

Here's another example of when you'd want to use Don't Find button: Suppose that in a multi-file search, you decide that you really want to set the Match Words option. Open the **Find** dialog, and change the option. If you were to click on the Find button, the search would go on to the next file. So you click on the Don't Find button, and continue using the **Find Again** and **Find in Next File** commands.

Disabling multi-file search

Entering a new search string cancels a multi-file search. To enter a new search string without cancelling a multi-file search, bring up the Find dialog. Click on the Multi-File Search check box. The list of all the text files will appear. Click OK to accept all the checked files, and then enter the search string in the "Search for" field.

To cancel a multi-file search without going through the multi-file selection dialog, hold down the Option or Command key as you click the Multi-File Search check box.

Finding the definition of a symbol

To find the definition of a symbol (a function name or a variable), hold down the Option (or Command) key as you double-click on it. THINK C opens the file in which the selected symbol is defined and looks for its first occurrence in that file. Usually, the first occurrence is the definition of the symbol. You can use the **Find Again** command if the first instance wasn't the definition. If the editor can't determine where the symbol is defined, you'll hear a beep. If the symbol is multiply defined, the editor arbitrarily opens one of the files that defines it.

This feature relies on information the compiler keeps in the project file, so the file that defines the symbol must be compiled. This feature only works for global (non-static) functions and variables.

Searching for a Pattern (Grep)

In addition to the search and replace functions described in the previous section, the THINK C editor also provides a powerful pattern search capability called Grep. The Grep search option in THINK C is based on the Grep utility on Unix systems. If you're familiar with this kind of pattern matching you'll find an old friend here. If pattern searching is new to you, experiment with this feature before you use it on a real file.

> **Note:** The editor will look for patters only when the Grep option is on.

Patterns

A pattern is a description of a set of strings rather than a specific string. For example, you can build a pattern that means "any word that begins with P." Or a pattern that means "any function call with `&event` as an argument."

Patterns can't span lines. So you can't write a pattern that means "three consecutive lines that begin with a, b, and c."

Simple Patterns

The simplest patterns match a single character.

- Any character, with the exceptions noted below, is a pattern that matches itself.
  Example: The pattern `2` matches a character 2. If you've checked Ignore Case in the **Find...** dialog box, any letter will match both its upper- and lower-case equivalent. So, either `a` or `A` will match both a and A.

- The character `.` is a pattern that will match any character.

- The character `\` followed by any character except `( ) < >` or one of the digits `1-9` is a pattern that matches that character.
  Example: `\.` matches a . and `\\` matches a `\`.

- A string of characters `s` surrounded by `[` and `]` is a pattern `[s]` that matches any one of the characters in the string `s`. The pattern `[^s]` matches any character that is not in the string `s`. If a string of three characters in the form `a-b` appears in `s`, this represents all of the characters from `a` to `b` inclusive. All other characters in `s` are taken literally. The only way to include the character `]` in `s` is to make it the first character. Likewise, the only way to include the character `-` in `s` is if it appears either at the beginning or at the end of `s`.
  Example: The pattern `[A-Za-z0-9]` matches any alphanumeric character. The pattern

`[^!-~]` matches any non-printing ASCII character. The **Ignore Case** option has no effect between brackets.

Complex Patterns

To match strings, not just individual characters, you need patterns that match consecutive sequences of characters. One way of doing this is to append a `*` to the end of one of the simple patterns.

- A pattern x followed by a `*` is a pattern x* that matches zero or more consecutive occurrences of characters matched by x.
  Example: The pattern `@*` matches a string containing any number of at-signs. If the string does not begin with an at-sign, or if it contains no at-signs at all, then the pattern matches the empty string at the beginning of the string to be matched. You'll see later on why this is useful.

You can put patterns together to form more complex patterns:

- A pattern x followed by a pattern y forms a pattern xy that matches any string ab, where a matches x and b matches y.
  Example: The pattern `P.` matches any string beginning with P and any other character.

- Of course, you can concatenate the compound pattern xy with another pattern z, forming the pattern xyz.

To put all of this together, consider the pattern `(.*)`. This pattern matches any string enclosed in parentheses. This includes the string `()`, since the sub-pattern `.*` will match the empty string between the (and the).

Will this pattern match the string `(())`? Since the sub-pattern `.*` will match any number of occurrences of all characters, won't the pattern match just the `(()` and not the very last)? The answer is any sub-pattern of the form x* in a pattern x*y matches the largest number of occurrences of whatever x matches that still allows a match to y. In matching `(())` against the pattern `(.*)`, only the inner pair of parentheses matches the sub-pattern `.*`, so the pattern will match `(())`.

Grep has a way of remembering sub-patterns so you can use them again as part of even more complex patterns. Things get a little complicated here.

- A pattern surrounded by `\(` and `\)` matches whatever the sub-pattern matches.
  Example: `\(a[b-y]z\)` matches the same thing as `a[b-y]z`.

- A \ followed by n, where n is one of the digits `1-9`, matches whatever the nth `\(` sub-pattern matched. You can add a `*` to a `\n` pattern to form a pattern `\n*` that matches zero or more occurrences of whatever `\n` matched.
  Example: To find two repeated words (like "the the") you might use a pattern like this:

`\([a-z][a-z]*\) \1`. This pattern matches a space, any sequence of letters, a space, and the same sequence of letters. Note that `\1` is not a reapplication of the pattern. Instead it becomes whatever the first `\(\)` pair matched.

Finally, you can constrain patterns to match only if they meet certain conditions in the context outside the string.

- A pattern surrounded by `\<` and `\>` matches whatever the pattern matches, provided that the first and last characters of the matched string match `[A-Za-z0-9_]` and that the characters immediately surrounding the matched string don't match `[A-Za-z0-9_]`. In other words, the pattern matches only if the string begins and ends on a word boundary. If you've checked Match Words in the **Find...** dialog box, the entire pattern you enter is treated as though it were surrounded by `\<` and `\>`.
  Example: To find occurrences of repeated words even if they're not surrounded by spaces, you would use the pattern `\(\<[a-z][a-z]*\>\)[^a-z]*\1`

- A pattern x preceded by a `^` forms a pattern `^x`. If `^x` is not preceded by any other pattern, it matches whatever x matches as long as the first character x matches occurs at the beginning of a line.

- A pattern x followed by a `$` forms a pattern `x$`. If the pattern `x$` is not followed by any other pattern, it matches whatever x matches as long as the last character that x matches occurs at the end of a line. If the pattern `x$` is followed by another pattern, then the `$` is taken literally.

These last two items constrain pattern matches to begin or end at line boundaries, and can be combined to constrain a pattern to match an entire line only.

Replacing with Grep

You can use Grep not only to search for strings, but also to replace them. The following special characters let you alter the replacement string.

- Each occurrence of the character `&` is replaced with whatever the entire pattern matched.
  Example: If you wanted to add a P to the beginning of every word that ended with ptr, you would search for `\<.*ptr\>` and replace it with `P&`.

- Each occurrence of `\n`, where n is one of the digits `1-9`, is replaced by whatever the nth occurrence of `\(` matched.
  Example: To change all strings like #define FOO 1 to FOO = 1, search for:
  `#define \(\<[A-Za-z0-9][A-Za-z0-9]*\>\) \(\<.*\>\)`
  and replace it with
  `\1 = \2`

- Each occurrence of a string `\x`, where x is not one of the digits `1-9`, is replaced by x.

Grep Examples

Grep is not easy to learn. To give you a hand, here are some typical examples .

Suppose that you've written a Macintosh application , and you've forgotten to put a `\p` at the beginning of your strings to signal to the compiler to make them Pascal strings rather than C strings. You can change all your C strings to Pascal strings by specifying

```
"\([^"]*\)"
```

as the search pattern and

```
"\\p\1"
```

as the replacement string.

To convert

```
symbol   equ   (expression+4)      ; a comment
```

to

```
#define symbol (expression+4)   /* ; a comment */
```

search for

```
\(\<.*\>\)[space tab]*\<equ\>\([^;]*\)\(.*\)
```

and replace with

```
#define \1 \2 /* \3 */
```

Explanation:

- `\<.*\>` matches a symbol.
- The surrounding `\(` and `\)` lets you use the symbol in the replacement string as `\1`.
- The `[space tab]*` matches any number of spaces or tabs between the symbol and the key word equ. (The words space and tab stand for the characters here because you can't see them on paper. To enter a Tab, type Command-Tab in the dialog box.)
- `\<equ\>` matches the word equ. It will not match equ if it is part of another word, for example equal. `\<equ\>` is not surrounded by `\(` and `\)` because it will be thrown away in the replacement string.

- `[^;]*` matches an expression formed by any number of characters up to but not including a ; (semi-colon).
- The surrounding `\(` and `\)` lets you use the expression in the replacement string as `\2`.
- The `.*` matches the comment which is the rest of the line.
- The surrounding `\(` and `\)` stores the comment as `\3`.
- If there was no ; (semicolon) in the line, then `\2` will consist of everything after the equ to the end of the line and `\3` will be an empty string.

To convert $HHHH to 0xHHHH, where H is a hexadecimal digit, Grep search for

```
$\([0-9A-Fa-f][0-9A-Fa-f]*\)
```

and replace with

```
0x\1
```

Explanation:

- `$` matches a `$`. `[0-9A-Fa-f]` matches one hex digit.
- `[0-9A-Fa-f][0-9A-Fa-f]*` matches one or more hex digits. (The pattern `[0-9A-Fa-f]*` matches zero or more hex digits.)
- The surrounding `\(` and `\)` lest you remember the hex digits in the replacement string as `\1`.

> **Note:** Save complicated Grep search and replace strings in a file so you can copy and paste them into the **Find...** dialog box.

Files & Folders
9

This chapter tells you why your files and folders are organized the way they are, how THINK C looks for #include files, and what happens when you move your files from one folder to another or to another machine.

Before you begin

Make sure that you followed the installation instructions in Chapter 2. If you didn't, go back now, and check to make sure that your disk is set up correctly. This chapter explains why your disk should be set up this way.

Topics covered in this chapter:

- Organizing your folders
- How THINK C names files
- How THINK C looks for #include files
- Moving files within a project
- Using the THINK C and project trees

Organizing Your Folders

This section tells you how THINK C knows where to find your source files, libraries, and #include files. THINK C expects to find your source files, libraries, and #include files relative to the folder THINK C is in or relative to the folder your project is in. The diagram at the end of this chapter shows you how your disk should be set up.

The THINK C and Project trees

THINK C treats all the files in all the folders within the `THINK C Folder` as if they were in the same flat folder. The `THINK C Folder` and all the subfolders in it are called the **THINK C Tree**.

THINK C treats all the files in your project folder as if they were all in the same folder. The folder and subfolders your project is in is called the **project tree**.

This organization lets you put your files into folders as you like, without worrying about pathnames. Since you use a standard file dialog to add files to your project, THINK C knows which tree they're in. If you move them around later on, THINK C looks in the appropriate tree to find where you put them.

How THINK C Names Files

When THINK C displays a file name in an edit window's title or in a **Get Info...** dialog, it uses these naming conventions to let you know which tree the file is in:

`<filename>`	THINK C Tree
`filename`	project tree

If a file isn't in the THINK C Tree or in the project tree, THINK C displays an absolute name for it:

`vol:filename`	top level folder
`vol:folder:filename`	subfolder of top level folder
`vol:…:folder:filename`	deeper subfolder
`vol:?:filename`	subfolder on unmounted volume

How THINK C Looks for #include Files

These are the rules THINK C uses to find #include files:

`<filename.h>`	THINK C looks only in the THINK C Tree
`"filename.h"`	THINK C looks first in the referencing folder, then in the project tree, and finally in the THINK C Tree.

The referencing folder is the folder that contains the file that has the #include preprocessor directive. For example, if a source file references an #include file MyUtils.h, and that file in turn has the line `#include "MyUtilTypes.h"`, THINK C will look for `MyUtilTypes.h` in the folder that contains `MyUtils.h` first.

Another example: You might use `#include <QuickDraw.h>` in your program to get the definitions for QuickDraw. If you look at `QuickDraw.h`, you'll see that it contains `#include "MacTypes.h"`. The `Mac #includes` folder is `QuickDraw.h`'s referencing folder, which is in the THINK C Tree, so that's where the preprocessor looks first to find `MacTypes.h`.

Once-only headers

You may want to create a header file that you want #included in several places but which should define its symbols only once in a project. THINK C makes this easy for you. It will not #include a header file if the symbol `_H_`*fname* is defined, where *fname* is the name of the file (in lowercase, without the `.h`). So, in the header file, simply define the symbol `_H_`*fname*.

For example, say you want the file `foo.h` to be #included only once in your project. In `foo.h` place the line:

```
#define     _H_foo
```

Wherever you want to #include the file, simply place the line:

```
#include    "foo.h"
```

You don't need check whether `_H_foo` is defined since THINK C does that for you.

Moving Files Within a Project

You can move files freely within a tree. If THINK C can't find a file that's already in your project, it assumes that you've just moved it somewhere within the tree, and tries to find it there.

Moving a source file

To move a file that's already in one of the trees to the other tree, it's best to use the **Save As...** command in the file menu. This command will take care of updating the references to the file in your project document without losing the object code, so you won't have to recompile it.

This is how you move a file from one tree to the other:

- Open the file you want to move. Double clicking on its name in the project window is the fastest way.
- Choose **Save As...** from the **File** menu. When the standard file dialog box appears, go to the folder within the tree you want to save the file in, and click on Save.
- Make sure you delete the file from the original tree. Since **Save As...** makes a copy, the original file is still in the old tree.

> **Note:** If you have Symantec's HFS Navigator™, you can delete the original file before you save it at the new location. When you get the standard file dialog box, hold down the Command key as you click on the folder pop up menu, and choose Get Info. Click on the delete button to delete the original file. (Don't worry, the file is in the edit window.) Then go on to save the file in the new folder.

Moving a library

Moving a library is a little trickier since you can't open libraries with the editor.

- Move the library to a folder in the other tree. Use the Finder or a file-moving desk accessory.
- Choose **Remove** from the **Project** menu to remove references to it from the project.
- Use the **Add...** command in the **Project** menu to put it back into the project.

Moving other files

From time to time your project may refer to files and libraries outside the THINK C or project trees. To move these files, use the same procedure as for libraries above.

Moving files to another machine

When you move a project to another Macintosh, use the Use Disk button in the **Make...** dialog to let THINK C find all the files. Files don't need to be in exactly the same place on the two machines as long as they're within the project or THINK C Trees. Files outside either tree must have the same absolute pathname on the two machines.

A note about search times

Searching the trees after you've moved files around can take a little time. Once THINK C finds a file, though, it remembers where it is and looks there first the next time. So if the compiler seems slower than usual, don't worry. Once THINK C learns where your files are, it will speed up again.

Using the Trees

The way THINK C keeps track of your files gives you the flexibility to organize your files just the way you like without having to specify full path names. There are a couple of points you should remember, though, about using the THINK C and project trees.

Don't put project folders in the THINK C Tree

This is the most common mistake. It seems natural to put all your THINK C files in one folder and then toss your project folders in there as well. If you set up your disk like this, THINK C will search *all* your other project trees every time it searches the THINK C Tree. Setting up your project folders this way not only increases search times, it also makes it more likely that you'll duplicate names within trees (see below).

Avoid duplicate file names in trees

Just as you can't have two files with the same name in the same folder, you shouldn't have duplicate file names in different folders within the project or THINK C Tree. If you do, THINK C won't know which file to use. Duplicate file names won't lead to any explicit errors, but you may end up using the wrong file.

It's OK to have the same file name in both the project and THINK C trees. THINK C resolves the conflict deterministically by search order.

> **Note:** The **Save As...** command copies files, so if you use it to save a copy in another folder, be sure to remove or rename the original file. See "Moving Files Within a Project" above.

Shielding folders from the trees

To shield a folder from the search tree, enclose its name in parentheses. For example, you might have a folder in the project folder named `(Backups)`. THINK C ignores all the files and sub-folders in shielded folders. Since THINK C doesn't see these files, it's OK if they have the same name as another file in the tree. Project specific folders are the only exception to this rule.

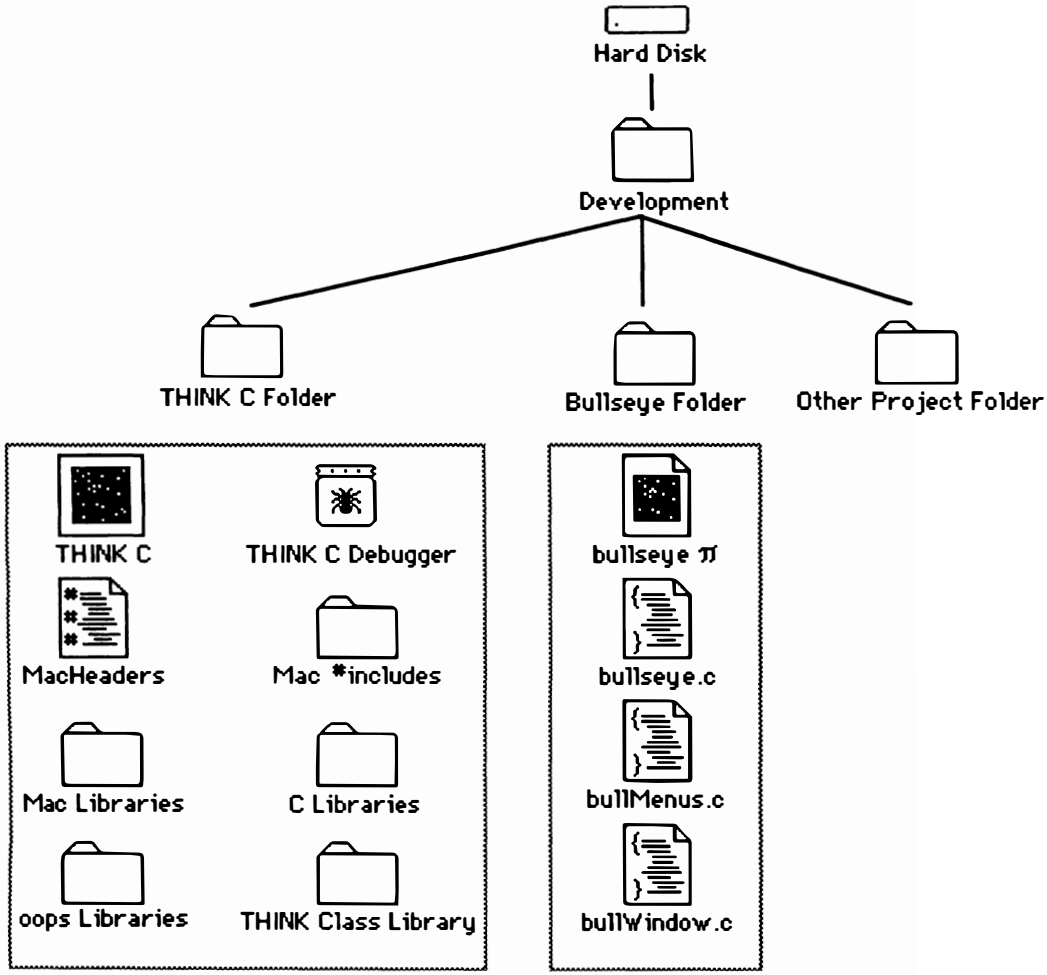
Project specific folders

There is one exception to the shielding rule above. If the folder your project is in contains a folder that has exactly the same name as your project surrounded by parentheses, THINK C *will* search that folder.

This feature is useful is you're working on two projects that share files. For instance, suppose you're working on two projects, `INITProject` and `DAProject`, that share some source files and are in the same folder. You create two folders, `(INITProject)` *and* `(DAProject)`, that both contain versions of the #include file `config.h` tailored to control conditional compilation of the common source files.

Disk Layout Diagram

This diagram shows the recommended disk layout. You don't have to set up your disk this way, but the important thing to remember is that your project folders should not be in the THINK C folder.



The Compiler
10

Introduction

This chapter describes features unique to the THINK C compiler. It tells you how to compile source files, how to use precompiled headers, and how to call the Macintosh Toolbox routines. This chapter also tells you how to set options that affect the way THINK C compiles your source files. If you're porting code from other compilers or writing code that will run on other machines, read the Portability section at the end of this chapter.

Topics covered in this chapter:

- Compiling source files
- Precompiled headers
- Calling Macintosh Toolbox routines
- Code generation options
- Compiler options
- Function prototypes
- Portability

Compiling Source Files

Unlike traditional compilers, THINK C doesn't generate separate **object files** from your **source files**. Instead, THINK C puts all the object code into the project document. Although you can compile files yourself, most of the time you'll be using the Auto-Make facility to compile your files.

> **Note:** Source files are your program files. Object code is the machine language that the THINK C compiler generates from your source files.

Compiling files not in the project

You can add a source file to your project and compile it in one step. First, create your source file with the THINK C editor. Save your file in the same folder as the project document. Make sure that the file name ends in `.c`. THINK C will only compile files that end in `.c`.

Next, choose **Compile** from the **Source** menu. A dialog box shows you how many lines THINK C has compiled. If there were no errors in the source file, THINK C adds the file and its object code to the project.

Compiling files already in the project

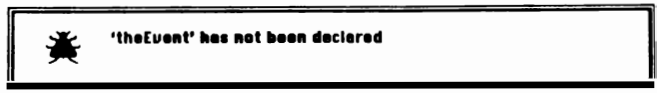
If you want to compile a file that is already in the project, just click on its name in the project window and choose **Compile** from the **Source** menu. Once a file is in the project, you don't need to open it to compile it.

Checking files without compiling

Sometimes you just want to make sure that your source file will compile without actually compiling it. The **Check Syntax** command in the **Source** menu checks the syntax of the contents of the frontmost edit window without generating code or adding the file to the project window. In fact, you don't even have to save the file first.

Fixing errors in source files

When THINK C detects an error in your source file, it opens the source file and displays a bug alert.



To get rid of this alert box, click anywhere in it or press the Return or Enter key.

The source file that contains the error will be in an editor window with the insertion point at the beginning of the line that contains the error.

Identifier length and capitalization

In THINK C every character in an identifier and its case is significant, even when the identifiers aren't in the same file. For example, suppose you had two files like this:

```
/*  file 1  */
int a_very_long_external_name;
---------------
/*  file 2  */
extern int a_very_long_external_nme;  /* misspelled */
```

THINK C would report the second misspelled identifier as an undefined symbol at link time.

THINK C is stricter than other compilers in this matter to help you catch spelling and capitalization errors.

Using register variables

You can declare up to eight register variables per function. Five variables can hold data in registers `D3-D7`, and three variables can hold addresses in registers `A2-A4`.

If the MC68881 option in the Code Generation section of the **Options...** dialog is on, THINK C uses floating point registers for variables declared `register double`. You can declare up to five floating point register variables. See "Code Generation Options" later on.

> **Note:** If you're building a desk accessory, device driver, or code resource, register `A4` is not available for register variables. These project types use `A4` to access their globals.

Floating point arithmetic

The THINK C type `double` corresponds to the SANE type Extended. This type takes up 10 bytes of storage, but is the *fastest* floating point type. Conversely, the THINK C type `float` corresponds to the SANE type Single. This type takes up 4 bytes of storage, and is the second fastest floating point type. The THINK C type `short double` corresponds to the SANE type Double. This type takes up 8 bytes of storage, but strangely enough is the slowest floating point type. If you want speed, stick with `double`. If you want to save space and are willing to sacrifice some accuracy and speed, then use `float`.

Precompiled Headers

THINK C lets you "precompile" header (#include) files. Precompiled headers contain only declarations and preprocessor symbols. Since precompiled headers are in a format THINK C can use readily, they load faster than text header files.

> **Note:** If you're using the source level debugger, you should use precompiled headers. Precompiled headers help make the debugger tables smaller.

THINK C comes with one precompiled header file, `MacHeaders`, which contains the most common declarations you use for writing Macintosh programs. When the MacHeaders option is on, THINK C automatically loads `MacHeaders`, so you never have to explicitly #include common header files like `QuickDraw.h`. (It doesn't hurt if you do, but it slows down compilation.) See "Code Generation Options" below to learn how to turn the MacHeaders option on and off.

The MacHeaders file contains these files:

```
ControlMgr.h
DeskMgr.h
DeviceMgr.h
DialogMgr.h
EventMgr.h
FileMgr.h
HFS.h
IntlPkg.h
ListMgr.h
MacTypes.h
MemoryMgr.h
MenuMgr.h
OSUtil.h
PackageMgr.h
QuickDraw.h
ResourceMgr.h
ScrapMgr.h
SegmentLdr.h
StdFilePkg.h
TextEdit.h
ToolboxUtil.h
WindowMgr.h
asm.h
pascal.h
```

These files aren't used as often, so they're not included in MacHeaders. You'll have to #include them yourself.

```
Appletalk.h
nAppletalk.h
Color.h
ColorToolbox.h
DeskBus.h
DiskDvr.h
FontMgr.h
MultiFinder.h
PrintMgr.h
ScriptMgr.h
SCSIMgr.h
SetUpA4.h
SerialDvr.h
SlotMgr.h
SoundDvr.h
SoundMgr.h
StartMgr.h
TimeMgr.h
Video.h
VRetraceMgr.h
```

Usually, you'll just use the built-in MacHeaders. You can, however, edit the default MacHeaders file or make your own precompiled headers.

Editing the MacHeaders file

You might find that in the kinds of program you write, you frequently refer to a header file that is not already in MacHeaders. Or, MacHeaders might include some files you never use. You can edit the MacHeaders file to suit the kinds of programs you write.

Open the file Mac #includes.c, and edit it to include any other files or declarations you need. Then, select **Precompile...** from the **Source** menu. After THINK C precompiles the file, save it as MacHeaders. (Precompiled files don't go into the project.) Be sure to place it in the same folder as THINK C.

The auto-make facility marks the files in the current project for recompilation if you change MacHeaders.

To let other projects know that `MacHeaders` has changes, use the **Make...**command in the **Source** menu to mark all the `.c` files, then click on the Make button to recompile *all* the files.

Creating your own precompiled headers

If you want to use your own precompiled headers, first disable THINK C's automatic loading of `MacHeaders`. Use the Code Generation radio button in the **Options...** dialog from the **Edit** menu to turn this feature off (see "Code Generation Options" below).

Create a file containing the desired series of #include statements, and choose the **Precompile...** command from the **Source** menu. When THINK C is through precompiling, save the file.

> **Note:** You can use `#include <MacHeaders>` as the first line of your precompiled header.

You use a precompiled header the same way you use any other header file. Use the `#include` statement to load it into your source file. The `#include` statement must be the first non-comment line of your source file. You can use only one precompiled header per source file. (If the MacHeaders option is enabled, you can't explicitly include any other precompiled header.)

When the MacHeaders option is off, you can use several different precompiled headers for different parts of your program, and you can still explicitly include `MacHeaders` if you want to use it in certain files.

> **Note:** You can use only one precompiled header per source file.

Calling the Macintosh Toolbox Routines

THINK C knows about all the routines in *Inside Macintosh I-V* including the ones marked `[Not in ROM]`. To use the Toolbox routines, call them exactly as they appear in Inside Macintosh. The only thing you need to know is how to convert the Pascal declarations into C declarations.

THINK C knows how many arguments every Toolbox routine takes and the sizes of the arguments. If you supply a wrong size integer (for instance, an `int` for a `long`), THINK C adjusts the argument automatically. If you pass the wrong size non-integer argument (an `EventRecord` instead of a pointer to an `EventRecord`, for example), THINK C displays an error dialog.

THINK C knows the sizes of the return values but not their types, so it assumes the values are integers. (Of course, for functions that do not return a value the return type is `void`.) The #include files supplied with THINK C contain declarations that specify the correct return type. For example,

```
extern pascal Handle GetResource();
```

ensures that the result from `GetResource` will be considered a `Handle`. Without this definition the result would be considered a `long`.

Most Macintosh calls are implemented by traps that use the Pascal calling conventions directly. THINK C generates these traps inline instead of generating a subroutine call. For register based traps or routines marked `[Not In ROM]`, THINK C generates calls to library functions in `MacTraps`. Calls to these routines also follow Pascal calling conventions.

Passing arguments to Toolbox routines

Since the argument declarations in *Inside Macintosh* are given in Pascal, you have to know how to convert them to C. You don't need to know the details of C and Pascal calling conventions to call the Toolbox routines, but it helps; see Chapter 12. This table shows you the general rule for converting argument declarations in Pascal to C:

If the object is...	**Pass...**
a `VAR` parameter	a pointer to the object
4 bytes or smaller	the object
larger than 4 bytes	a pointer to the object

Here are some examples of Pascal parameter declarations and their C counterparts:

Pascal Type	**C Type**
`INTEGER`	`int`
`LONGINT`	`long`
`CHAR`	`int`
`BOOLEAN`	`char`
`Byte`	`Byte` in `struct` declarations `int` when passed as an argument.
`VAR Byte`	`int *`
`ProcPtr`	See "Pascal callback routines" below
`Handle`	`Handle`
`VAR Handle`	`Handle *`
`Ptr`	`Ptr`
`VAR Ptr`	`Ptr *`
`OSType, ResType`	`long`
`PACKED ARRAY [1..4] OF CHAR`	`long`
`String255`	`String255` or `char *`

Pascal Type	C Type
`VAR String255`	`String255` or `char *`
`StringPtr`	`StringPtr` or `char *`
`VAR StringPtr`	`StringPtr *` or `char **`
`Rect`	`Rect *`
`VAR Rect`	`Rect *`
`Point`	`Point`
`VAR Point`	`Point *`

Toolbox routines that take strings as arguments expect them to be Pascal strings. Unlike null-terminated C strings, Pascal strings begin with a length byte. To write a Pascal string constant, start your string with `"\P"` or `"\p"`. This is how you would call the QuickDraw routine `DrawString()`:

```
DrawString("\pThis is a Pascal string");
```

Because Pascal strings start with a length byte, the largest Pascal string is 255 bytes.

> **Note:** Pascal strings are *not* null terminated.

Use the routines `CtoPstr()` and `PtoCstr()` convert back and forth from Pascal strings to C strings. These routines convert the strings in place, and return the converted string. These are their function prototypes:

```
char *CtoPstr(char *s);
char *PtoCstr(char *s);
```

> **Note:** If you turn the MacHeaders option off, be sure to #include `pascal.h` when you use these functions.

Some Macintosh Toolbox routines use the Pascal type `PACKED ARRAY[1..4] OF CHAR` to specify resource types and file types. Although this is an array type in Pascal, don't treat it as an array in C. Since it is only 4 bytes long it is passed by copying it onto the stack, and should be declared in C as a `long`. THINK C lets you specify multi-byte character constants like `'STR#'` or `'TEXT'` for this purpose.

The QuickDraw data type `Point` is only 4 bytes long, so it's passed by copying it onto the stack. Don't pass its address unless it is a `VAR` parameter. Most `RECORD` (`struct`) types, however, are larger than 4 bytes, so you would have to pass their address.

Special cases of Toolbox routines

Because the Macintosh Toolbox is written in Pascal, some functions work differently when you call them from THINK C. Fortunately, the number of cases is very small.

The `Fix2X()` and `Frac2X()` functions (see *Inside Macintosh IV*, Chapter 12, "Toolbox Utilities") are defined as returning `Extended`. In THINK C, these functions return `double`. These two functions are the only Toolbox functions that use C calling conventions. They are not declared `pascal`. See `ToolboxUtil.h` for their declarations.

The List Manager function `LLastClick()` returns `long` instead of `Cell` as documented in *Inside Macintosh IV*, Chapter 30, "The List Manager."

The "old" File Manager data structures described in *Inside Macintosh II*, Chapter 4, "The File Manager" are defined in `FileMgr.h`. The "new" data structures described in *Inside Macintosh IV*, Chapter 19, "The File Manager" are defined in `HFS.h`. The "new" definitions have an `H` prepended to the data structure name. For instance, the "new" `VCB` structure is called `HVCB`.

The routines `SetUpA5()` and `RestoreA5()` described in *Inside Macintosh II*, Chapter 13, "The Operating System Utilities" are provided as macros instead of as functions in `MacTraps`. They are defined in `OSUtil.h`.

Using the RAM serial driver routines

In the 64K ROMs, your application needs a `SERD` resource to use the RAM serial driver described in *Inside Macintosh II*, Chapter 9, "The Serial Drivers." This resource was incorporated into later ROMs. If your application uses the serial driver, and you want it to run on 64K ROM machines, you'll find the `SERD` resource in the file `SERD` in the Mac Libraries folder. Use ResEdit to copy the resource into your application's resource file.

Calling AppleTalk routines

If your application uses AppleTalk, you should be aware that there are two sets of AppleTalk interfaces. Apple calls the old interface (described in *Inside Macintosh Volume II*) the **alternate** set. The new interface (described in *Inside Macintosh Volume V*) is called the **preferred** set. You can use either or both sets of interfaces.

To use the alternate AppleTalk routines, use the library `AppleTalk` and the #include file `AppleTalk.h`. To use the preferred routines, use the library `nAppleTalk` and the #include file `nAppleTalk.h`.

To use this interface...	**Use this library / header**
preferred	`nAppleTalk / nAppleTalk.h`
alternate	`AppleTalk / AppleTalk.h`

Calling Print Manager routines

There are two ways to use the Print Manager. If you want your project to run under System Software version 4.1 or earlier, #include `PrintMgr.h` and add the library `PrGlue`. This way, calls to Print Manager routines will use the Print Manager traps directly. If your project

will run on System Software later than version 4.1, #include `PrintTraps.h` and add the library `MacTraps`.

To run under...	Use this library / header
System Software 4.1 or earlier	`PrGlue / PrintMgr.h`
System Software later than 4.1	`MacTraps / PrintTraps.h`

Using SANE utilities

To use certain utility functions from the Standard Apple Numerics Environment (SANE), add the `SANE` library to your project and #include the file `SANE.h` in your files. This library implements such functions as `x80tox96()`, `x96tox80()`, `str2dec()`, `cstr2dec()`, `dec2str()`, and more. For more information on SANE, read *Apple Numerics Manual, Second Edition* (Addison-Wesley).

The SANE library uses 80-bit values and is not intended for projects that have the MC68881 Code Generation option checked. If that option is checked, be sure to use the `x80tox96()` and `x96tox80()` functions as necessary.

These eight functions are defined by both the `SANE` and `ANSI` libraries:

```
atan()                  log()
cos()                   sin()
exp()                   sqrt()
fabs()                  tan()
```

To use the `SANE` versions, #include the file `SANE.h` before the file `math.h`. Similarly, to use the `ANSI` versions, #include the file `math.h` before the file `SANE.h`.

Pascal callback routines.

Some Macintosh Toolbox routines take a pointer to another function as an argument. Those routines then call the function you passed in. The function you provide is called a **callback** routine. The Toolbox routines expect the callback routines to follow Pascal calling conventions. (To learn about the difference between Pascal calling conventions and C calling conventions, see Chapter 13.)

THINK C gives you a way to write functions that behave as though they were written in Pascal. The function definition must begin with the `pascal` keyword. Make sure you provide a return type. If you're writing a function that behaves like a Pascal `PROCEDURE`, the return type is `void`.

For the parameter declarations, follow the same rules as for calling Toolbox functions. Remember that non-`VAR` parameters are supposed to be passed by value, not by reference. If the size of a non-`VAR` parameter is greater than 4 bytes, you'll need to pass its address, but you may not modify the parameter.

For example, `ModalDialog()` lets you provide a `filterProc` to handle events in your dialog. This is how *Inside Macintosh* declares `ModalDialog()`:

```
PROCEDURE ModalDialog (filterProc: ProcPtr; VAR itemHit: INTEGER);
```

`ModalDialog()` expects the `filterProc` to have this declaration:

```
FUNCTION MyFilter (theDialog: DialogPtr; VAR theEvent: EventRecord;
VAR itemHit: INTEGER) : BOOLEAN;
```

In THINK C, the declaration for `MyFilter()` would look like this:

```
pascal Boolean MyFilter (DialogPtr theDialog, EventRecord *theEvent,
int *itemHit)
```

The call to `ModalDialog()`, then, looks like this:

```
extern pascal Boolean MyFilter();
int theItem;

ModalDialog(MyFilter, &theItem)
```

Keeping C and Pascal on speaking terms can be tricky, but THINK C tries to make it as painless as possible.

Calling Pascal routines indirectly

Only Macintosh Toolbox routines and functions that are declared `pascal` are called using Pascal conventions. If you have a pointer to a Pascal function, you can call the function by using one of the library functions `CallPascal()`, `CallPascalB()`, `CallPascalW()`, or `CallPascalL()`.

For example, suppose that `pC` is a pointer to a C function and `pP` is a pointer to a Pascal function; both functions accept two `int` arguments and return `void`. You can declare both pointers the same way:

```
void (*pC)(), (*pP)();
```

To call the function pointed to by `pC` you would write:

```
(*pC)(5, 7);
```

But to call the function pointed to by `pP` you write:

```
CallPascal(5, 7, pP);
```

Note that the pointer to the function must be the last argument to `CallPascal()`. When you use these routines, be aware of the different return types.

> **Note:** If you turn the MacHeaders option off, be sure to #include `pascal.h` when you use these functions.

To call a Pascal...	**Use...**
`PROCEDURE`	`CallPascal(arg1, arg2, ..., fp)`
`FUNCTION returning BOOLEAN`	`CallPascalB(arg1, arg2, ..., fp)`
`FUNCTION returning INTEGER, CHAR`	`CallPascalW(arg1, arg2, ..., fp)`
`FUNCTION returning LONGINT, Ptr, Handle`	`CallPascalL(arg1, arg2, ..., fp)`

Accessing low memory globals

The #include files (or `MacHeaders`) define most of the low memory globals referenced in *Inside Macintosh*. THINK C provides a way to define additional low memory globals:

```
extern int MemErr : 0x220;
extern char FinderName[] : 0x2E0;
```

The address you provide cannot be greater than 0xFFFF.

Defining inline functions

You can define inline functions using this form:

returnType functionName (*arguments*) = { *instr1*, *instr2*, ... };

THINK C will replace a call to the function *functionName* with the machine instructions *instr1*, *instr2*, ... and *not* with a function call. For example, the function `PrOpenPage()` can be defined as follows:

```
pascal void PrOpenPage() = { 0x2F3C, 0x1000, 0x0808, 0xA8FD };
```

Now THINK C will replace a call to `PrOpenPage()` with these machine instructions:

```
move.l  #0x10000808, -(SP)
_PrGlue
```

This lets you create interfaces to the Macintosh Toolbox that are compiled as efficiently as the ones that are built into the compiler. Some of THINK C's header files, such as `PrintTraps.h`, use this format. Look at them for more examples.

Defining MC68881 unary inline functions

When the MC68881 Code Generation option is enabled (from the Code Generation section of the **Options...** dialog), you can declare unary inline functions for the MC68881 as *follows*:

returnType functionName (*argument*) : *instr;*

where *instr* is the low six bits of the second word of the desired MC68881 instruction.

> **Note:** This syntax works with unary functions (i.e., functions with only one argument) only.

For example, this is the inline function definition for `atanh()`:

```
double atanh (double) : 0x0D;
```

The header file `math.h` defines some unary functions in this way. You may want to define others yourself.

Tips

`SetRect`, `SetPt`, etc., take more time than setting up the coordinates yourself because of the overhead of the Macintosh trap dispatcher. However, you'll notice significant improvements only if you are executing lots of them.

If you are dereferencing handles, make sure that the memory the handle points to won't move while using it. Otherwise, `HLock` it (and `HUnlock` it later). Example:

```
HLock(aTEH);
/*  Without the previous HLock, */
/* the next two calls may   not always work*/

/*(1) printf calls Memory Mgr: */
printf("first character in text handle is %c\n", **(**aTEH).hText );

/*(2) Handle dereference on the left is done BEFORE call:*/
(**aTEH).hText = (Handle)NewHandle(somesize);

HUnlock(aTEH);
```

To increase your heap zone (the memory available to your program) to the maximum, call the procedure `MaxApplZone()`.

Don't forget to make all necessary Macintosh initialization calls. In THINK C, you must make all the necessary calls yourself.

You don't need to call initialization routines when you use the `ANSI` library. Initializations are automatically done the first time an `ANSI` procedure (that needs it) is called. You can turn this off if you're doing your own initializations — for example, if you are using the standard output window for debugging. The automatic initialization is for users of THINK C who want to have the more traditional, non-Macintosh C environment.

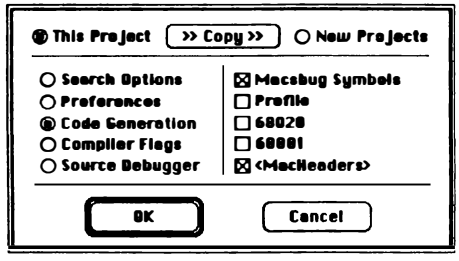
The `Byte` and `Char` types in Pascal actually correspond to the `int` type in C. The three places that this causes problems are the functions `ATPOpenSocket()`, `GetItemMark()`, and `GetItemIcon()`. In these functions, a `Byte` argument is passed by reference. The proper way to do this in C is to pass a pointer to an `int`.

Code Generation Options

THINK C lets you control some aspects of code generation. The simplest way you can affect code generation is by using `register` variables. You can also use the **Options...** command in the **Edit** menu to instruct THINK C to generate symbols for assembly language debuggers, to generate special code for the code Profiler (see Appendix A), and to generate code for the MC68020 CPU or the MC68881 coprocessor. You can also use the same dialog to tell THINK C whether to use the precompiled `MacHeaders` (described above).

Setting the options

To set the code generation options, choose **Options...** from the **Edit** menu. Click on the Code Generation radio button, and you'll see this dialog box:



Chapter 55 talks about the **Options...** command in detail.

Using Macsbug symbols

When the Macsbug Symbols options is set, THINK C generates symbols for assembly language level debuggers such as Macsbug or TMON . THINK C generates symbols only for functions that have stack frames, so functions without arguments and local variables don't get symbols. Be aware that while Macsbug symbols are useful for debugging, they add 8 bytes to every procedure. This option is on by default.

Using the Profile options

When the Profile option is set, THINK C generates calls to the code profiler routines. The code profiler collects timing statistics about your functions. See Appendix A to learn more about the code profiler. This option is off by default.

Generating MC68020 code

If the MC68020 option is checked, THINK C uses the MC68020 instructions for bitfield operations and long word multiplication, division, and modulo operations. Also, the inline assembler accepts MC68020 addressing modes that are not valid for the MC68000. (See Chapter 12, "Assembly," for more information).

> **Note:** Before using MC68020 instructions, be sure to check that the code is running on a machine with a MC68020. The MC68020 code will crash a machine with a MC68000. (You can use the Toolbox routine `SysEnvirons()` to find out which processor your application is running on.)

Generating MC68881 code

If the MC68881 option is checked, THINK C generates inline code for the floating point coprocessor. Up to five local variables of type `double` may be declared `register` and will be placed into MC68881 registers.

When this option is on, the size of double variables is 96 bits. Since SANE expects 80 bit doubles, you'll need to convert from one format to the other. The `float` (32 bits) and `short double` (64 bits) types are identical for SANE and for the MC68881.

If you have this option set, you'll need to use the MC68881 version of the math library.

> **Note:** Before using MC68881 instructions, be sure to check that the code is running on a machine with the chip. The MC68881 code will crash a machine without the chip. (You can use the Toolbox routine `SysEnvirons()` to find out what coprocessors your application's machine has.)

Using the MacHeaders option

When the <MacHeaders> option is on, THINK C will automatically include the `MacHeaders` file for every file in your project. The `MacHeaders` file contains the declarations for the most common Macintosh Toolbox types, functions, and low memory globals. Since these declarations are in binary form, compilation is faster than if you included the header files manually. It doesn't hurt to include header files already included in `MacHeaders`, like `QuickDraw.h`, but compilation will be a bit slower.

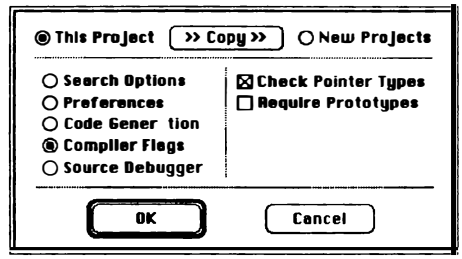
If you want to use your own precompiled headers, make sure this option is turned off. THINK C allows only one precompiled header per source file.

To learn more about precompiled headers, see the Precompiled Headers section in this chapter.

Compiler Options

THINK C lets you specify how strictly it should enforce type checking. You can set the compiler options to make sure that THINK C checks that pointer types are assignment compatible. You can also enforce some strict type checking by requiring that every function have a function prototype. (To learn about function prototypes, read the next section.)

To set the compiler options, choose **Options...** from the **Edit** menu, and click on the Compiler Flags radio button. You'll see this dialog box:



Chapter 55 talks about the **Options...** command in detail.

Check Pointer Types

When the Check Pointer Types option is on, THINK C makes sure that pointer types match when you assign one pointer to another or when you do pointer arithmetic. If this option is off, THINK C treats all pointers as equivalent types, and won't display the "pointer types do not match" error message. When subtracting two pointers, however, the two types must be pointers to objects of the same size.

Require Prototypes

When the Require Prototypes option is on, THINK C forces very strict type checking: You can't use or define a function unless it has a prototype. If this option is on, even functions with ANSI-style definitions need prototypes. Read the next section to learn about function prototypes.

> **Note:** Macintosh Toolbox routines and the function `main()` never need prototypes, even if the Require Prototypes option is on.

Function Prototypes

A function prototype is a function declaration with additional information that describes the arguments the function takes. These function declarations are also function prototypes:

```
int strcpy(char *dest, char *source);
extern int printf(char *, …);
```

You can specify functions with a variable number of arguments with ellipsis (...) as in the `printf` example above. The ellipsis must appear at the end of the argument list. No type information is provided about the additional arguments. Argument identifiers are optional and are ignored if supplied.

A function declaration with no argument information, e.g.

```
long foo();
```

is *not* a prototype and supplies no information about the arguments. It only declares the return type. To write a prototype specifying that the function takes no arguments, use

```
long foo(void);
```

> **Note:** This is a special case and does *not* mean that the function takes a `void` argument.

Prototypes are optional unless you have checked the Require Prototypes option. If you use prototypes, they must appear before the first definition or use of the function. The best place for prototypes is in #include files, so other files in your project will be aware of them.

If a prototype is in effect when a function is called, the actual arguments are checked against the prototype. Unless the prototype ends in ellipsis (...), the number of arguments must match exactly. The argument types must be assignment-compatible (the state of the Check Pointer Types option is honored). Appropriate conversions are applied to arithmetic (integral and floating-point) values. Additional arguments allowed by the ellipsis are not checked or converted.

If no prototype is in effect when a function is called or defined, the "null" prototype (...) is assumed. However, if the Require Prototypes option is checked, THINK C displays an error message. (THINK C never requires prototypes for the Macintosh Toolbox routines, but you can supply them if you prefer.)

You can supply prototype information for a Macintosh Toolbox routines. THINK C checks the prototypes against the built-in size and argument count information, and will subsequently use the prototype in preference to the built-in information. For example:

```
extern pascal Handle GetResource(OSType, int);
```

Full prototype information is not built in for the Macintosh calls.

An old-style function header is not itself a prototype, so the following example won't work:

```
f(x)
long x;
{
    ...
}
g()
{
    ...
    f(0);
    ...
}
```

The 0 argument is passed to `f()` as an `int`, not a `long`. To get automatic type coercion, you must supply this prototype before you call the function `f()`:

```
int f(long);
```

Portability

The American National Standards Institute (ANSI) has published a standard definition of the C language. To learn more about the ANSI definition of C, see the second edition of Kernighan and Ritchie 's *The C Programming Language* (Prentice Hall) or Brodie and Plauger's *Standard C* (Microsoft Press).

New language features in this release make THINK C more compatible with the ANSI C standard. Although THINK C is close to the standard, it is not conformant as defined in the standard. The remaining issues either are of little significance or would require fundamental changes to THINK C. For more information on ANSI compatibility, read the Chapter 57.

Predefined preprocessor symbol

THINK C predefines a preprocessor symbol, `THINK_C`, so you can test for THINK C when doing conditional compilation.

Sizes of numbers

The most common way C compilers differ is in the sizes of the three integer types, `short`, `int`, and `long`. There seems to be general consensus that:

1. `short` should be the shortest integer type bigger than char supported by the hardware
2. `long` should be the longest integer type supported by the hardware
3. `int` should be the "most natural" integer type supported by the hardware

On the MC68000 this means that `short` should be 2 bytes and `long` should be 4 bytes, and most compilers do it this way. The correct size for plain `int` is more problematic. The MC68000 has 32-bit registers, so some compilers (including MPW C) implement 4-byte `int`s. But the MC68000 has a 16-bit data bus and a 16-bit ALU, so 16-bit operations are considerably more efficient than 32-bit operations. Furthermore, Macintosh applications are often pressed for memory, and 2-byte `int`s use a lot less space than 4-byte `int`s, so some compilers implement 2-byte `int`s. THINK C implements 2-byte `int`s.

If you are porting from a compiler which has different integer sizes, and you have code which relies on those sizes, you'll have to do some conversion. The most common case is code that assumes that `int` and `long` are the same size. Here is an example:

```
foo()
{
    long a, b, result;
    result = baz(a, b);
}
baz(i, j)
{
    return(i + j);
}
```

This code works fine when `int` and `long` are the same size, but it is not portable. It won't work in THINK C (or in many other compilers). Either `i` and `j` should be declared `long`, or `a` and `b` should be cast to `int` before being passed to `baz`.

It is less common for a program to make specific assumptions about the sizes of floating-point numbers, but you should be aware that these also tend to differ between compilers. For maximum accuracy and efficiency, THINK C uses the SANE or MC68881 extended-precision type to implement the C type `double`. Other compilers for the Macintosh either don't provide the extended-precision type at all, or add a new largest type.

Passing a Point as an argument

Because the QuickDraw `Point` is a structure, some compilers require that you pass the address of a `Point` instead of the `Point` itself. They require, for example,

```
result = PtInRgn(&aPoint, aRgnHandle);
```

even though `PtInRgn` expects the actual `Point` as its first argument.

In THINK C, the above statement would cause `aPoint`'s address to be passed to `PtInRgn`; the correct call is:

```
result = PtInRgn(aPoint, aRgnHandle);
```

Some Toolbox functions expect a `Point` to be passed as a `VAR` parameter; in such cases an address must be passed. For example,

```
SetPt(&aPoint, horizontal, vertical);
```

would be correct in THINK C as well as in other compilers.

Converting from Unix

THINK C comes with a complete implementation of the standard ANSI libraries. To use any of them, add the `ANSI` library to your project and #include the appropriate header files. See the *Standard Libraries Reference* for more information.

THINK C also supports many UNIX features that aren't in the ANSI libraries. For example, the `ccommand()` function lets you emulate a UNIX command line. Again, see the *Standard Libraries Reference* for more information.

The Debugger
11

This chapter describes THINK C's source level debugger. The source level debugger lets you debug your application the way you wrote it: in C. The debugger lets you step through your program line by line, set breakpoints, examine and set the values of your variables.

This chapter describes some of the more advanced features of the THINK C debugger. Chapter 5 is a tutorial that teaches you how to use the debugger. You might want to work through that tutorial before you read this chapter.

Before you begin

Make sure the file `THINK C Debugger` is in the same folder as THINK C. If you followed the installation instructions, it should be in the folder named `THINK C Folder`.

To use the source debugger you need at least 2Mb of memory, and you must be running THINK C under MultiFinder. The debugger works only with application projects. It won't work with code resources or device drivers. To use the source debugger to debug desk accessories, use the file `DA main.c` in the disk `THINK C 1` and follow the instructions there.

Topics covered in this chapter:

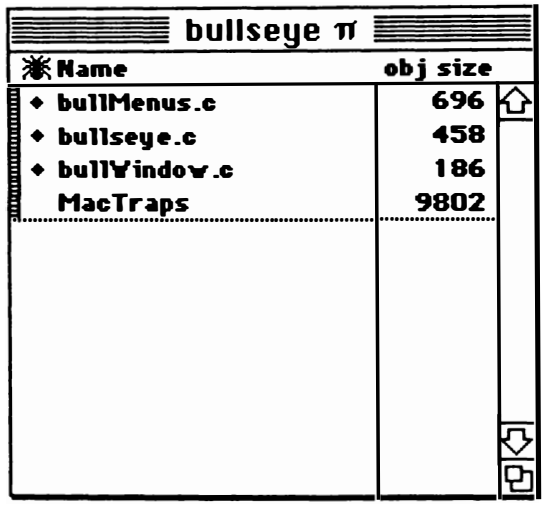
- Running with the debugger
- The debugger windows
- Working with the Source window
- Setting breakpoints
- Controlling execution
- Working with the Data window
- Using low level debuggers
- Quitting the debugger
- *Memory considerations*

Running with the debugger

The debugger runs as a subordinate application with THINK C. Although there is a debugger document icon, you can't launch it from the Finder by itself. When you set the option to use the source debugger, THINK C takes care of launching it. Make sure that the `THINK C Debugger` file is in the same folder as THINK C.

Turning the debugger on

To run your application with the debugger, choose the **Use Debugger** command in the **Project** menu. This command turns the Use Debugger option on and off. When the option is on, THINK C adds a "bug" column to the project window.



THINK C generates debugging information for files that have gray diamonds next to them. Initially, all files get gray diamonds. THINK C never generates debugging tables for libraries or projects used as libraries.

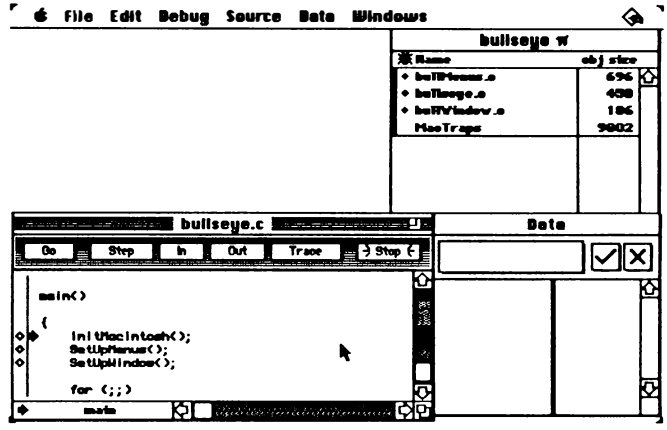
Clicking in the "bug" column next to a file name turns the gray diamonds on and off. If you hold down the Option key as you click on a gray diamond, THINK C removes the gray diamonds from every file. If you hold down the Option key as you click in the "bug" column where there isn't a gray diamond, THINK C turns the diamonds on for every file in the project.

> **Note:** Another way to turn the debugger on is to check the Use Debugger check box in the Source Debugger section of the **Options...** dialog.

Running the project

When you run your program, THINK C launches the source debugger instead of launching your program. The debugger then gets ready to run your program.

The debugger displays its own menu bar and two windows at the bottom of your screen. If you're using a Macintosh II with two screens, and you have the Use 2nd Screen option on, the two debugger windows appear in the second screen.



The larger window on the left is the **Source window**. It contains the debugger status panel and the source text of your program. The window on the right is the **Data window**. Use the Data window to display and change the values of your variables.

When your application is running, the screen shows the application's menu bar, and its windows are brought forward. When your application stops (at a breakpoint, for example), the debugger's menu bar appears, and its windows are brought forward.

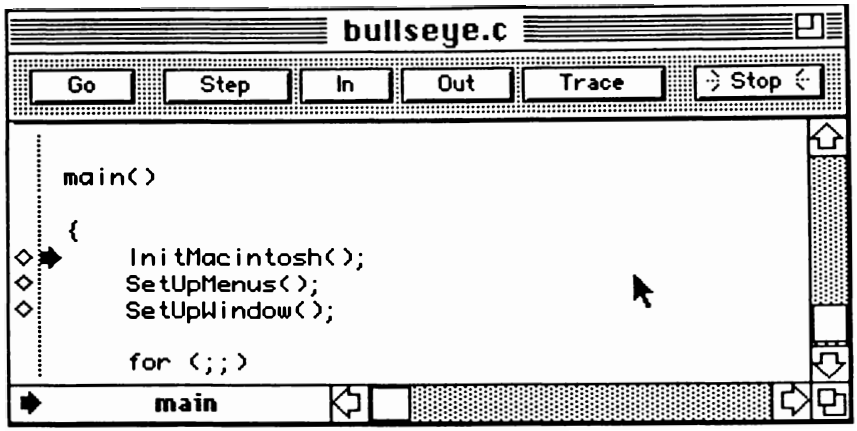
> **Note:** Unlike other Macintosh applications, the debugger doesn't use the **File** menu, so it's always dimmed. When your screen is cluttered with a lot of debugger and application windows, the dimmed **File** menu tells you instantly that the debugger's windows are active.

The Debugger Windows

The debugger windows show you the source of your program and the values of your variables. Since the debugger works like any other application running under MultiFinder, you'll need to be aware whether its your application or the source debugger is in the foreground. Just because the debugger windows are frontmost doesn't mean that your program isn't running. In fact, if your application can run in the background under MultiFinder, it will continue doing its background processing.

The Source window

The Source window contains the source text of your program, the debugger's status panel, statement markers, the current statement arrow, and the current function indicator. The title of the source window is the name of the source file.



The Source window shows the **source text** of your program. When you start the debugger, this window shows the file that contains the `main()` routine of your application. See "Working With the Source Window" to learn how to see other files in your project in the source window.

The top of the Source window has a six button **status panel**. These buttons control the execution of your program. The names of these buttons match the commands in the debugger's **Debug** menu. See "Controlling Execution" below to learn how to use the status panel and the **Debug** menu commands to step through your code.

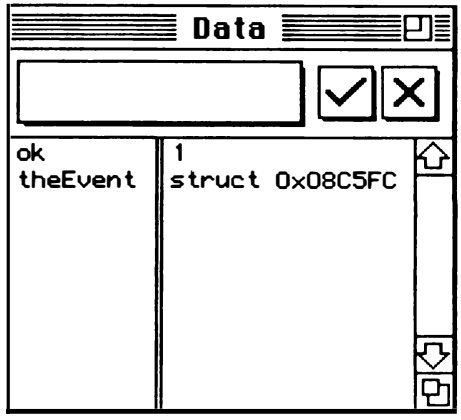
The column of diamonds running along the left side of the source text are **statement markers**. Every line of your program that generates code gets a statement marker. You can set breakpoints only at statement markers. See "Setting Breakpoints" to learn how to set and clear breakpoints.

The black arrow to the right of the statement markers is the **current statement arrow**. This indicator shows you the **current statement**, the one that the debugger is about to execute. When you first start your program, the current statement arrow is at the first executable line of your program.

The source debugger uses the the space at the lower left of the source window for the name of the **current function**. When you click here and hold the mouse button down, the debugger displays a pop-up menu that shows the call chain — the names of the functions that were called to get to the current function.

The Data window

The other debugger window is the Data window. In this window you can examine and set the values of your variables.



The Data window is modeled after a spreadsheet. Expressions you type into the **entry field** appear in the left column when you press the **enter button** (check mark) or when you press the Return or Enter key. Pressing the Enter key leaves the expression selected. Pressing the Return key leaves the entry field empty so you can type in the next expression. The enter button works just like the Enter key.

If you change your mind and don't want to enter an expression, press the **deselect button** (X mark). The expressions you enter appear in the left column and their values are displayed in the right column.

You can select items in either the expression column or in the value column. If it's legal to edit the item, you can use the entry field to edit the item.

You can drag the center bar to make a column wider.

To clear an expression in the Data window, select it and choose **Clear** from the **Edit** menu or press the Clear key.

Working with the Source Window

The debugger gets the text for the source window directly from your source file, so you see the file exactly the way you wrote it. As you step through your code and move from file to file, the debugger gets the text for the files it needs. The Source window is read only. You can select text in to copy, but you can't change it.

The debugger displays the source text only if THINK C generated debugging information for it. If you step into a function that's in a file that didn't have a gray diamond next to it in the project window, the debugger displays the message `Debugging information not available`.

If the debugger can't find a source file as it was when it was last compiled, the debugger displays the message `Source text not available`. (See "Editing while debugging" below.) This will happen, for instance, when the debugger can't find a source file or when a source file has been edited.

The current statement and the selected statement

The **current statement** is the one that the debugger is about to execute. The current statement arrow always points to the current statement.

Some of the debugger commands require you to select a statement in the Source window. To select a statement, just click once anywhere on its line. This is called the **selected statement**.

> **Note:** All commands that work on the selected statement operate on the current statement if you didn't select any other statement.

If the current statement arrow isn't visible (because you've scrolled away or because you're viewing another source file in the Source window), click on the current function indicator in the lower left of the source window, or press the Enter key. The source file that contains the current statement will appear in the Source window.

The current function indicator

The current function indicator at the lower left of the Source window displays the name of the function that the current statement is in. If the current statement is in a library, the current function indicator displays the name of the file. If the current statement is in ROM, the current file indicator displays the address of the program counter.

When you click and hold the mouse button down on the current statement indicator, you'll see a pop-up menu that shows you the **call chain**. The call chain follows stack frames from register `A6`, so if a function doesn't generate a stack frame, you won't see it in the call chain.

If you choose an item in the call chain pop-up menu, the debugger opens the file that the selected function is in, and selects the line that the called function would return to.

Viewing other files in the source window

The Source window usually shows the file that contains the current statement. To look at another file in the Source window (to set breakpoints in it, for example) you tell THINK C to send the text to the debugger:

- Click on the THINK C project window (Or choose your project name from the **Windows** menu)
- Click on the name of the file you want to look at
- Select **Debug** (Command-G) from THINK C's **Source** menu

The source debugger's Source window will display the text of the file you chose. Now you can examine the file and set breakpoints in it. To get back to the current file, click on the current function indicator at the lower left of the Source window or press the Enter key.

Editing files while debugging

Because THINK C is still running, you can edit your source files while you're debugging. To edit the file that's in the source window, choose the **Edit *'filename.c'*** command in the **Source** menu.

To edit any other file in your project, click on the project window (or choose your project name from the **Windows** menu), and edit your files normally.

Naturally, the changes you make to source files won't take effect until you recompile.

Whenever the debugger needs the source text for a file, it looks for it on disk. The debugger will cache the file as long as it can, but if it needs the memory for something else, it may reclaim the cache space. If you edit and save a file that the debugger has displayed in the Source window, it will continue to display the unedited version of the file as long as it's still cached. If the debugger sees that the source file on disk and the object code in the project don't match, it displays the message `Source text not available.` in the Source window.

Searching in the Source window

To search for something in the source window, you use the THINK C editor.

- Choose **Edit 'filename.c'** from the **Source** menu (or press Command-E) to open an edit window for the file in the Source window.
- Use the THINK C **Find...** command to find what you're looking for.
- Choose **Debug** from the THINK C **Source** menu (or press Command-G) to get back the source debugger. The line containing the selection in the editor window is displayed in Source window.

When the Source window displays what you're looking for, you'll usually want to use the **Go Until Here** command. See "Controlling Execution" later on to learn about this command.

Setting Breakpoints

The THINK C debugger lets you set breakpoints at any line that has a diamond statement marker. When your program is running, the debugger stops execution just before a breakpoint.

You can set three kinds of breakpoints: simple breakpoints, conditional breakpoints, and temporary breakpoints. Execution always stops at a simple breakpoint. Conditional breakpoints let you use the value of an expression in the Data window to determine whether execution should stop. Temporary breakpoints let you set a breakpoint and start execution in a single step. When your program reaches a temporary breakpoint, execution stops, and the debugger automatically clears the temporary breakpoint.

You can set breakpoints while your program is running, not just when it's stopped.

Simple breakpoints

To set a breakpoint, click on a statement marker diamond. The diamond will turn from hollow to filled, indicating that the breakpoint is set.

To clear a breakpoint, click on the filled diamond. The statement marker changes to a hollow diamond to show you that the breakpoint is clear. The **Clear All Breakpoints** command in the **Source** menu clears all the breakpoints.

> **Note:** You can also use the **Set Breakpoint** and **Clear Breakpoint** commands in the **Source** menu to set and clear breakpoints. Select a line in the source window by clicking once on the line. Then choose **Set Breakpoint** or **Clear Breakpoint** from the **Source** menu.

Setting breakpoints in other files

The Source window usually shows the file that contains the current statement. To set breakpoints in another file, you must first tell THINK C to send the file to the debugger:

- Click on the THINK C project window (Or choose your project name from the **Windows** menu)
- Click on the name of the file you want to look at
- Select **Debug** (Command-G) from THINK C's **Source** menu. The source debugger's Source window will display the text of the file you chose.
- Click on statement marker diamonds to set breakpoints.

To get back to the current file, click on the current function indicator at the lower left of the Source window or press the Enter key.

Setting temporary breakpoints

To set a temporary breakpoint, hold down the Command or Option key as you click on a statement marker. When you release the mouse button, the debugger starts your program, and execution continues until you hit any breakpoint, not just the temporary breakpoint. The source debugger clears all the temporary breakpoints when you stop for any reason.

Temporary breakpoints are useful if you want to go quickly to a particular line of your program. For instance, suppose you wanted to examine how your program handled menu selections. You'd set a temporary breakpoint at the first line of your menu handling routine. The program would run normally until you chose a command from one of your menus.

The **Go Until Here** command (see "Controlling Execution" below) sets a temporary breakpoint at the selected line.

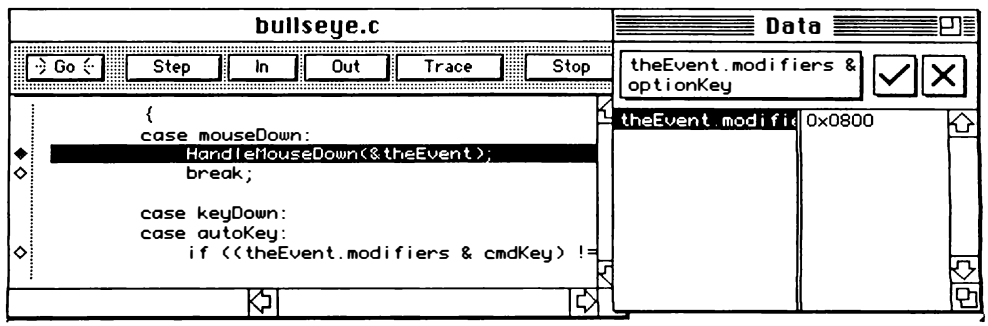
Setting conditional breakpoints

The THINK C debugger lets you set conditional breakpoints. A conditional breakpoint is a breakpoint has a condition associated with it. The debugger stops execution at conditional breakpoints only when the condition evaluates to non-zero.

To set a conditional breakpoint:

- Set a breakpoint by clicking on the statement marker diamond
- Click on the line to select it
- Click on an expression in the Data window
- Choose **Attach Condition** from the **Source** menu.

When you set a conditional breakpoint, the statement marker turns to a gray diamond.



In this example, the debugger will stop at the breakpoint only if you hold down the Option key as you click the mouse (`theEvent.modifiers & optionKey == 0x0800`).

If you're already stopped at a simple breakpoint that you want to turn into a conditional breakpoint, just select an expression in the Data window, and choose **Attach Condition**.

Since the debugger uses the current statement if there isn't a selected line, the condition will be attached to the current statement.

The **Attach Condition** command will be dimmed if:

- the expression can't be evaluated in the context of the breakpoint
- a breakpoint isn't set
- an expression in the Data window isn't selected
- the expression is a floating point expression

As you debug your program, you may forget the condition that's associated with the breakpoint. To see the condition associated with a conditional breakpoint:

- Click on the line to select it
- Choose **Show Condition** from the **Source** menu.

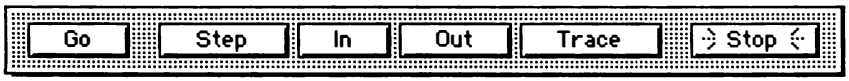
The condition associated with the conditional breakpoint will be selected in the Data window.

To learn how to use the Data window, see "Examining Variables" below.

Controlling Execution

The debugger has six commands that control execution. To make it easier to debug your programs, there are three ways to use them: you can choose them from the **Debug** menu, you can use the Command key equivalents, or you can use the buttons in the status bar of the Source window.

The buttons in the status panel do double duty as status indicators. When your program is running, the Go button is lit. When your program is stopped, the Stop button is lit. Remember that your program can still be running even if the debugger windows are frontmost.



Go

The Go command starts your program if it was stopped. Your program will run until you stop it (with the Stop button, for example) or until it's about to execute a line with a breakpoint or until it hits an exception (dividing by zero, for instance). If your application is already running, the Go command brings it to the foreground.

Trace

The Trace command executes the current statement. In most cases, the statement indicator will go on to the next statement marker, even if the next statement marker is in another function. The only time it won't is when the program counter steps into some code that the debugger doesn't have the source text for. This usually happens when you step into a trap that's not generated inline. So, for a brief period, the current statement arrow isn't really anywhere in your program, but somewhere in `MacTraps` instead. Though you can't see the current statement arrow, the current function indicator at the lower left tells you which file it's in.

Step

The Step command is like the Trace command except that it goes on to the next statement marker in the current function. If you're at the end of a function, Step returns to the calling function. Use Step when you want to follow the execution within a function without falling into another function. (Technically speaking, Step skips over `JSR`s.)

Step In

The Step In command executes Trace commands until the current statement arrow falls into a function. Step In is useful when you want to skip over a set of assignments or Toolbox traps, to fall into the next function call. If Step In reaches the last statement of the current function without falling into another function, it will stop immediately after the function returns.

Step Out

The Step Out command executes Step commands until the current statement arrow falls out of the current routine. This operation can be slow if there's a lot left to do, but it's a sure way of leaving the current routine. A faster way of leaving the current routine is to use the **Go Until Here** command or to set a temporary breakpoint at the last diamond in the function.

Stop

The Stop command stops execution of your program. The Stop command works when any debugger window is active, and the Stop button only works when the source window is the active window. When you press the Stop button you'll usually be coming out of your call to `GetNextEvent()` or `WaitNextEvent()`.

If your program is not in its event loop, you might not be able to make the debugger the frontmost application. In this case, Command-Shift-Period is the **panic button**. Use Command-Shift-Period to stop execution when one of your application's windows is frontmost or when you think it's stuck in a loop. (Command-Shift-Period won't work if you're stuck in an infinite loop in ROM, though.)

Go Until Here

The **Go Until Here** starts execution and stops at the selected line. This command is exactly the same as setting a temporary breakpoint (see "Setting Breakpoints" above) at the selected line. Use this command when you want to move through a block of code quickly.

This command is more convenient than setting a temporary breakpoint when the line you want to go to is already selected. For instance, it's easier to press Command-H after you've found a string you're looking for. See "Searching in the Source window" above.

Skip To Here

The **Skip To Here** command changes the program counter to the selected line without executing any intervening code.Use it when you want to skip over code you know to be buggy but not crucial to the rest of the program's operation.

> **Note:** This command is potentially dangerous. Make sure the code you're skipping to doesn't depend on anything the skipped code does. For instance, it is be a very bad idea to skip over initialization routines.

Stepping continuously

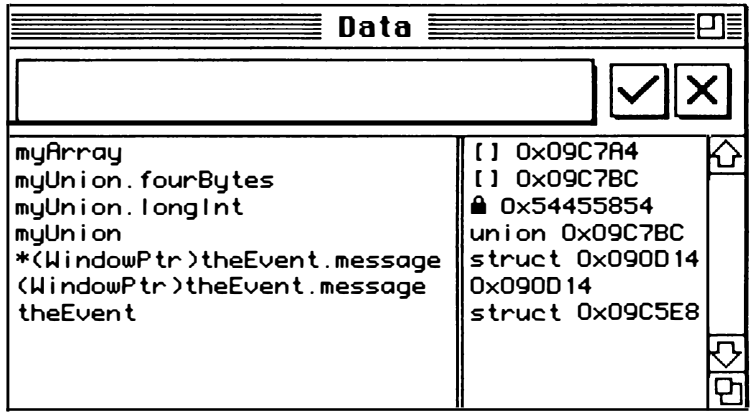
If you hold down the Option or Command key as you click on one of the status panel buttons (except Stop) you'll enter **auto mode**. In auto mode the debugger updates the Source and Data windows and repeats the command when the program stops. To trace through every line of your program automatically, for example, hold down the Option key as you click on the Trace button.

One useful technique is to set breakpoints at spots where you'd like to look at some variables and then do an Auto-Go. When your program hits the breakpoint, the debugger will update the Data window and start the program up again.

To cancel auto mode, type Command-Shift-Period or click on the Stop button in the status panel. The debugger will finish the command and then stop. Note that if you were doing an Auto-Go, the Stop button just cancels auto mode. You'll have to click on the Stop button again to stop your program.

Working with the Data Window

The Data window lets you examine and modify the values of your variables as you debug your programs. You can type any legal C expression in the left column, and its value is displayed in the right column. You can display values in several formats to make your debugging easier. You can also display structs and arrays in their own windows.



Entering an expression

You can enter any expression that does not have a potential side effect. That means you cannot enter assignment statements, function calls, or expressions involving ++, -- , +=, etc.

The debugger compiles the expressions you enter in the context of the selected line in the Source window, or, if you haven't selected a line, within the context of the current statement. If the expression you enter is not defined within the context, the debugger will display an error message as the value of the expression.

Expressions in the Data window have either local scope or global scope. An expression has local scope if it refers to variables with dynamic storage; in other words if it refers to non-static variables local to a function. All other expressions have global scope.

If you use the Enter key to enter an expression, the expression remains selected. If you use the Return key to enter an expression, the entry field is ready to accept the next expression. Clicking on the enter button is the same as pressing the Enter key.

Editing expressions

To edit an expression in the Data window, select it in the left column. The expression will appear in the entry. Use standard Macintosh editing techniques to edit the text of the expression.

When you edit an expression in the data window, the context is the same as when you entered the expression. To change the context of an expression to the current context after you edit it, hold down the Option or Command key as you click on the enter button.

Formatting values

When you enter an expression, it's added to the left column. The value of the expression appears in the right column. You can use the formats in the **Data** menu to change how the debugger displays the variables.

To change a format, select an expression in the Data window. Then choose a format from the **Data** menu. Some of the formats won't be available. The formats available depend on the type of the expression. The default formats are shown in italics.

Type	**Formats Available**
integers	*decimal*, hex, char
unsigned	*hex*, decimal, char
pointers	*pointer*, address, hex, C string, Pascal string
arrays	*address*, C string, Pascal string
structs	*address*
unions	*address*
functions	*address*
floats	*floating point*

This is what the display formats look like:

Format	**Example**
decimal	`4523345, -23576`
hex	`0xA09E1487`
char	`'c', 'TEXT'`
C string	`"abcdef\nghi\033"`
Pascal string	`"\pabcdef\nghi\033"`
pointer	`0x7A7000`
address	`[] 0x09FE44, struct 0x08FC14`
floating point	`1961.0102`

The C string and Pascal string formats display non-printing characters in backslash form. Whenever it can, the debugger uses the built-in escape characters (`\n`, `\r`, `\b`); otherwise it uses `\nnn`, where nnn is an octal value.

Of course, you can use type casting to use formats that aren't normally available. For example, if you really wanted to see an integer, `i`, as a C string, you would type this expression: `(char *) i`.

To see any pointer as an array, just change its format to Address. This way, when you double-click on its value, you'll see an array window instead of the value of what the pointer points to.

Displaying and changing contexts

If you forget the context of an expression in the Data window, use the **Show Context** command in the **Data** menu. The Source window will display the context for the expression.

When you select an expression, you can edit it in the entry field. When you press the enter button (or the Return or Enter keys), the debugger will recompile the expression in its original context.

To change the context of an expression you've already entered or edited, select the context in the Source window, select the expression, then choose **Set Context** from the debugger's **Data** menu. You can do the same thing by holding down the Option or Command key as you click on the enter button (or press the Return or Enter key).

Evaluating expressions

The debugger re-evaluates the expressions in the data window every time your program stops. Expressions whose context isn't in the current function are not re-evaluated unless they have global scope. To keep the Data window from getting too cluttered, the debugger clears their values.

Sometimes, you don't want an expression to be re-evaluated. For instance, you might want to compare the values of the same expressions at different times. Select an expression and choose **Lock** from the **Data** menu.

Setting values

The debugger lets you change the values of your expressions as long as the expression would be legal in the left side of an assignment statement.

Working with expressions

Double-clicking in the left column of the data window makes a copy of the expression. This is useful if you want to lock one copy down while you let another be evaluated every time the debugger stops.

Double-clicking on the value of struct, union or array opens up a new window.

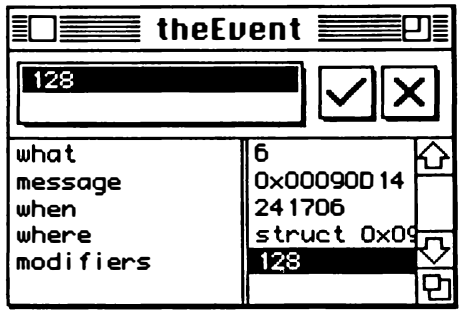
Double-clicking on a value formatted as Pointer enters a new, dereferenced, expression in the Data window. If you hold down the Shift key, the new expression will be dereferenced twice.

If you hold down the Option key as you double-click, the new entry replaces the original entry.

Examining structs and arrays

You can examine fields of structs and arrays displayed in the Data window. (In this section, whatever you read about structs applies to unions as well.)

When you double-click in the right column on an expression whose value is struct, the debugger opens up a struct window. The struct window looks like the data window. The names of the fields appear in the left column, and their values appear in the right column.



You can change the values of the fields of the records the same way you change any variable. Of course, you can't change the names of the fields.

Double-clicking on a value opens other windows the same way as double-clicking in the main data window. A new expression appears in the main Data window for the new window.

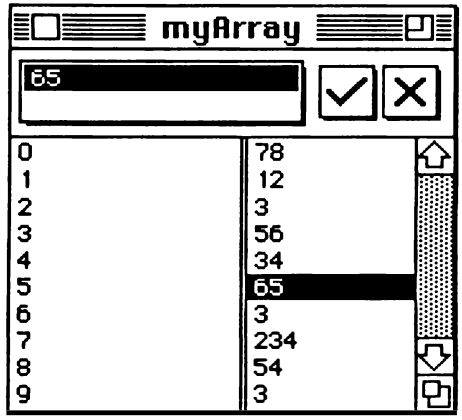
> **Note:** All struct and array windows "belong" to the main Data window. For every struct or array window there is an entry in the main Data window.

Double-clicking on a field name enters a new expression in the main Data window.

As mentioned above, Option-double-clicking replaces the original expression with the new one. For example: Suppose you had an struct window for an `EventRecord`, `theEvent`. Option-double-clicking on the `what` field would make the struct window disappear (the original expression is `theEvent`), and `theEvent.what` would appear in its place in the Data window.

Array windows are similar to struct windows. The indices appear in the left column and values appear in the right column. Unlike the main Data window or struct windows, every element in an an array is displayed in the same format.

Because C compilers don't enforce array bounds, array windows have "infinite" scroll bars.



If you select an index in the left column and change its value, the debugger will display the array from that index.

To see a pointer as an array, set its format to Address and double-click on its value.

Using Low Level Debuggers

The THINK C source level debugger is great for figuring out what your application is doing. But sometimes you need to get closer to the machine. The **Monitor** command in the **Debug** menu invokes a low level debugger like TMON or Macsbug.

When you use the **Monitor** command, all registers and low memory globals contain the correct values. The PC (program counter), however will not be pointing to the next instruction. Instead, it will be pointing somewhere in the debugger.

> **Note:** If you don't have a low level debugger installed, don't use the **Monitor** command.

Using the Monitor command with TMON

If you're using TMON, the value of the PC will be in TMON's V register. It's a good idea to keep a TMON disassembly window anchored to V so you can see your program's code when you use the Monitor command.

If there was an expression or value selected when you dropped into TMON, the N register contains that value.

To return to the THINK C debugger, use TMON's Exit command.

Using the Monitor command with Macsbug

If you're using Macsbug, the correct value of the PC is right before the current PC. To display your program's code in Macsbug, type:

```
DM PC-4
IL @.
```

The value of the current Data window selection is not available in Macsbug.

Use Macsbug's G command to return to the THINK C debugger.

Leaving the low level debugger

If you entered the low level debugger through the Monitor command, use your debugger's standard exit function: Exit in TMON, G in Macsbug.

If you got into your low level debugger any other way, there is no simple way to return to the source debugger. You can use your low level debugger's ExitToShell to abort your program as long as it's the foreground application. Check the low memory global `CurApName` (`0x0910`). If it contains the name of your program, go ahead.

Quitting the Debugger

The best way to quit the debugger is to quit your application. You should use the **ExitToShell** command in the debugger's **Debug** menu only when you can't use your application's **Quit** command.

Memory Considerations

THINK C, the source debugger, and your project each run in their own MultiFinder partition. The default partition sizes are enough to ensure that you can run all three with enough space left over for the Finder and the System on 2Mb of memory.

The default partition sizes are:

Application	Partition Size
THINK C	700K
debugger	200K
project	384K

If you want to run other applications, or if you find you're running out of memory, you can change the partition sizes. Change your project partition first. If you still don't have enough memory, change the size of the THINK C partition. Finally, change the debugger's partition.

When you're running under MultiFinder, your application doesn't need to reserve memory for the system heap, so you can set the project partition size to be quite small. For moderate size projects you might want to try values like 128K.

THINK C uses the most memory when its compiling. If THINK C complains that it's running out of memory when you compile, close any open windows before rebuilding your project. The THINK C partition should be no smaller than 500K.

If you have the Update Windows debugger option on (Source Debugger options under the **Options...** command in the **Edit** menu), the debugger partition might need to be bigger than 200K. You will definitely need to increase it if you're running with either large or color monitors. If this option is off, and you're running a small project, you can make the debugger partition as small as 150K.

To change the size of your project's partition, use the **Set Project Type...** command in the **Project** menu. You can change the partition sizes for THINK C and the debugger from their **Get Info...** boxes in the Finder. You can't change the partition size of an active application, so you'll have to quit THINK C first.

Assembly Language
12

Introduction

This chapter tells you how to use assembly language in your THINK C programs. You can use THINK C's built in inline assembler, or you can use object files generated by other assemblers. This chapter also describes C and Pascal calling conventions so you can write well-behaved assembly code.

What you should know

You should know the assembly language for the MC68000 family of microprocessors. Motorola publishes these references for their processors: the *MC68000 8-, 16-, 32-Bit Microprocessors User's Manual, Sixth Edition*, the *MC68020 32-Bit Microprocessors User's Manual, Third Edition*, the *MC68030 Enhanced 32-Bit Microprocessor User's Manual*, the *MC68851 Paged Memory Management Unit User's Manual, Second Edition*, and the *MC68881/882 Floating-Point Coprocessor User's Manual, Second Edition*.

Topics covered in this chapter:

- Using the inline assembler
- Differences from other assemblers
- C calling conventions
- Pascal calling conventions
- Tips

Using the Inline Assembler

THINK C lets you use assembly language within your source files. The THINK C **inline assembler** works within the compiler to produce object code. You can use instructions for the Motorola MC68000 and MC68020 processors and for the MC68881 floating point coprocessor. You can refer to C variables and functions within assembly language routines. Your C routines can `goto` labels in the assembly routines and vice versa.

The `asm` keyword invokes the inline assembler. The syntax for mixing assembly language statements with C code is simple:

```
asm {
    ...        /* assembly instructions, one per line */
}
```

The compiler treats this construct as a C statement. It can appear anywhere a C statement can appear, which means that it must appear within a function.

Use only one assembly language instruction per line. You can use semicolon style comments, but since you're still writing in C, you can use C style comments as well. Preprocessor symbols and macros are expanded within assembly language, just as they are within C. You can use a C constant expression wherever a constant would be legal.

The inline assembler supports all the standard MC68000, MC68020, and MC68881 (also called the floating point unit) instructions and addressing modes. THINK C does not yet support the assembly language extensions for the PMMU (also called the MC68851), and you can not use the PMMU instructions available on the MC68030.

> **Note:** Using only MC68000 assembly language ensures that your code will run on all Macintosh computers.

The inline assembler follows assembly language syntax conventions with a few exceptions. The DC (define constant) directive, which places literal values in the code stream, is the only assembly language directive the inline assembler recognizes. Use C to declare data, to define symbolic constants, and to import and export symbols.

The assembler is not case sensitive with respect to instruction mnemonics, register names, and size specifications (`.B`, `.W`, `.S`, `.L`, `.P`, `.X`, and `.D`).

Writing MC68000 assembly

There are two ways to use MC68000 assembly. Most of the time, you'll want to uncheck the MC68020 option (in the Code Generation section of the **Options...** dialog) and enclose the assembly in an `asm { ... }` construct.

However, when the MC68020 Code Generation option is checked, you may want to ensure that a certain piece of assembly code is interpreted using only MC68000 addressing modes and not with the MC68020 extensions. Enclose your assembly in this construct:

```
asm 68000 {
    ...         ; MC68000 instructions.
}
```

> **Note:** These two methods will not let you use MC68020 addressing modes, but they will let you use MC68020 instructions. If you want to use MC68020 instructions, however, you should follow the instructions below in "Writing MC68020 assembly" to be sure your code will compile properly in future releases of THINK C.

Writing MC68020 assembly

There are two ways to use MC68020 assembly. Most of the time, you'll want to check the MC68020 option (in the Code Generation section of the **Options...** dialog) and enclose the assembly in an `asm { ... }` construct.

However, you might want to use MC68020 extensions in a project that has the MC68020 option unchecked. For example, you might want to use an extension that helps your application run faster , you want to ensure that your application will still run on the Macintosh Plus and SE (albeit more slowly). In this case, you can enclose the MC68020 assembly in this construct:

```
asm 68020 {
    ...        ; MC68020 instructions.
}
```

> **Note:** Before using MC68020 instructions, be sure that the code is running on a machine with a MC68020 or MC68030. The MC68020 code will crash a machine with a MC68000. You can use the Toolbox routine `SysEnvirons()` to find out which processor your application is running on.

Writing MC68881 assembly

The inline assembler will accept all MC68881 instructions and addressing modes, even when the MC68881 Code Generation option is disabled (from the Code Generation section of the **Options...** dialog). In fact, the the MC68881 Code Generation option does not affect the inline assembler.

You may enclose your MC68881 assembly in the following construct to make it easier to identify, but the construct doesn't affect how the code is interpreted:

```
asm 68881 {
    ...    ; MC68881 instructions
}
```

> **Note:** Before using MC68881 instructions, be sure to check that the code is running on a machine with the chip. The MC68881 code will crash a machine without the chip. You can use the Toolbox routine `SysEnvirons()` to find out what coprocessors your application's machine has.

You may use valid C floating point literals as immediate-mode operands to MC68881 instructions. When you need to specify an immediate operand that is accepted by MC68881 but that is not a valid C floating point literal, enclose the operand in quotes. For example:

```
fmove.x #"INF",fp2
    ; "INF" stands for the MC68881 representation of infinity.

fmove.x #"123456678901234567890",fp1
    ; Without the quotes, the assembler would attempt to store
    ; the number as an integer and cause an overflow. With them,
    ; it stores the number as a double float.

fmove.x #123456678901234567890.0,fp1
    ; The ".0" also forces the assembler to store the number
    ; as a double float.
```

Using C identifiers in assembly language

The inline assembler lets you use C identifiers directly. The base register (`A6` for locals, `A5` or `A4` for globals, `PC` for labels) is optional, but must be correct if supplied.

If you've declared register variables, you can use their names wherever a register would be legal. The inline assembler is case sensitive with respect to C identifiers, just like the C compiler.

> **Note:** C identifiers which conflict with register names (`D0-D7`, `A0-A7`, `SP`, `USP`, `SR`, `CCR`, etc.) cannot be referenced by name in inline assembly.

If you declare a variable to be stored in a register and THINK C can't store it in one, the assembler will signal an error when you use the variable in a register context. For example, this code fragment could cause problems without this restriction:

```
foo ()
{
    register long   *a, *b, *c;
    ...
    asm {
        move.l  (c), d0
        ...
    }
}
```

If `c` is placed in a register, the `move.l` instruction would fetch the long word `c` points to. But if `c` can't be placed in a register (for example, if this were part of a code resource, where only two address register variables are available) the `move.l` instruction would fetch the value of `c` itself. Because of the restriction, however, the assembler would signal an error in

the second case. Note that if c wasn't used in a register context, the assembler wouldn't signal an error.

You can reference fields of a `struct` directly. For example, if you have a variable of type `WindowRecord`, you can get the `refCon` field like this:

```
long myRefcon() /*  refCon of myWindow  */
{
    extern WindowRecord myWindow;
    asm {
        move.l  myWindow.refCon,d0
    }   /*  same as "return(myWindow.refCon)"  */
}
```

Use the `OFFSET()` macro in the #include file `asm.h` to get offsets of fields of a structure. For example, if you only had a pointer to a `WindowRecord`, you'd use the `OFFSET()` macro to get the `refCon` field:

```
long refcon (WindowPtr wp)  /*  same as "GetWRefCon"  */
{
    asm {
        move.l  wp,a0
        move.l  OFFSET(WindowRecord,refCon)(a0),d0
    }
}
```

Labels and branching

You can label assembly language instructions with C or assembler labels. An assembler label consists of an at sign (@) followed by one or more alphanumeric characters. Colons are optional following assembler labels, but must appear after C labels.

```
foo ()
{
    asm {
    @123    ...   /* A legal asm label */
    @2_px:  ...   /* A legal asm label */
    here:   ...   /* A legal C label   */
    }
}
```

The scope of an assembler label is the enclosing function, not just the sequence of inline assembly in which it appears. This is the way C labels work, too.

Note: A short branch (`BRA.S`) to the immediately following instruction is an error which is *not* detected by the inline assembler. (This instruction generates an 8-bit zero displacement, which results in the next instruction word being used as a 16-bit displacement for a long branch rather than being executed as an instruction.) To work around this, use a word branch (`BRA.W`) or a branch without a size qualification (`BRA`). See the *Motorola MC68000 User's Manual* for more details.

You can use these C branching statements within assembly language:

```
break       ; exits the surrounding loop or switch
continue    ; skips to next iteration of the surrounding loop
return      ; exits function
goto label  ; same as "bra @label"
```

You can `goto` C labels in assembly language from C code.

You can also refer to C labels from inline assembly, whether the label appears in assembly code or in C code. The label must be preceded by `@` to indicate to the assembler that it is a label. (This avoids ambiguity in statements such as: `lea foo, A1`.)

The assembler optimizes branch instructions without size qualifications (such as `BNE`), choosing whether to generate a long or a short branch. This optimization does not apply to instructions with size qualifications, such as the `BNE.S` or `BNE.W` instructions.

The assembler also optimizes unconditional branches around C statements appearing within inline assembly. For example,

```
while (...) {
    ...
    asm {
        bne @1
        continue
    @1  ...
        ...
    }
}
```

generates the same code as

```
while (...) {
    ...
    asm {
        beq @cont
    @1  ...
        ...
    }
cont:
}
```

As above, this optimization applies only to instructions without size qualifications; for example, it apples to the BNE instruction, but not the `BNE.S` or `BNE.W` instructions.

Do not use the RTS instruction unless you're absolutely sure you know what you're doing. Use `return`, or simply fall through to the end of the function. This way, the C stack frame will be cleaned up properly.

If you use the `return` statement, don't specify a return value. Put the return value in the proper place (usually D0) instead. See the following sections about C and Pascal calling conventions for more information.

If you JSR to a function from within assembly language, the function must be already declared. For example, to call the function `MyFun()` from assembly:

```
extern int MyFun();

OtherFun()
{
    ...
    asm {
        ...
        jsr MyFun
        ...
    }
}
```

It's up to you to make sure that you pass the correct arguments for a function you call from assembly language. See the sections below for C and Pascal calling conventions.

Multiple entry points

You can create multiple entry points into a function by using labels declared as `extern`. First, declare each additional entry point to be a new function. Then, inside an inline assembly block, mark the beginning of each additional entry point with a label preceded by

extern. The names of the label and the new function must be the same. For example, this code fragment creates an additional entry point into `foo()` called `foo1()`:

```
void foo1();

void foo()
{
    asm {
            moveq   #0,d0
            bra.s   @1
    extern foo1:
            moveq   #1,d0
    @1      ...
            ...
    }
    ...
}
```

> **Note:** Use this feature carefully! It is very powerful, but improper use can wreak havoc.

Using the Macintosh Toolbox in assembly language

You can use any of the Macintosh Toolbox functions (except those marked `[Not in ROM]`) in your assembly language routines. The inline assembler is case sensitive with respect to trap names. If you like, you can precede trap names with an underscore.

The inline assembler generates one instruction for ROM traps even though some might require glue when you call them from C. If you want to use the glue from assembly language, you'll need to JSR to the routine. For example:

```
DemoFun()
{
    asm {
    /* Direct, register based call */
        MOVE.L  #256,D0
        _NewHandle
        /* result in A0, error code in D0 */
        ...
    /* Stack based call through glue */
        CLR.L   -(SP)       /* save space for result */
        MOVE.L  #256,-(SP)  /* push number of bytes */
        JSR NewHandle
        /* Result at (SP), error in MemErr */
    }
}
```

Note: With the 64K ROMs, Memory Manager traps do not set the low-memory global `MemErr`, though the glue does. This is a time when you'd want to use the glue instead of using the trap directly.

You can provide an optional argument, a 2-bit value to be placed in bits 9–10 of the trap to set trap modifier bits, such as `AUTOPOP`, `SYS`, `CLEAR`, and `ASYNC`. The various trap modifier bits are defined in `asm.h`. For example:

```
Handle NewClearSysHandle(size)
{
    asm {
        move.l   size,d0
        NewHandle    CLEAR+SYS
        move.l   a0,d0
    }
}
```

Register usage

You may modify registers `D0`, `D1`, `D2`, `A0`, and `A1`, as well as registers holding register variables. All other registers should be saved and restored as you need them. If you want to use a register other than for scratch purposes, declare a register variable. You'll be able to refer to it by name, and you won't have to bother to save and restore its value.

If intervening C code is executed between two stretches of inline assembly, you can assume that the C code preserves the values of registers `A5`, `A6`, and `A7` — as well as `A4` for drivers and code resources. All other registers may have been modified. It is safe to leave things on the stack while C code is executing, provided you clean up the stack before returning from the function.

See the next sections for more information about C and Pascal calling conventions.

Differences from Other Assemblers

This sections describes the differences between THINK C's inline assembler and most other assemblers.

Hexadecimal constants

THINK C does not accept the syntax `$NNNN` to designate a hexadecimal constant. Use the C syntax `0xNNNN` instead.

THINK C treats hexadecimal constants as unsigned numbers. For example, `0xFF` is equivalent to `255`, `0xFFFF` is equivalent to `65535`, and neither is equivalent to `-1`.

Order within operands

THINK C has no rules specifying the order of registers and constants within assembly operands. For example, these statements are legal:

```
tst.w   2(a0,d0)     ; Allowed by most assemblers.
tst.w   2(d0,a0)     ; Disallowed by most assemblers,
                     ;   but allowed by THINK C.
```

There is one instance in which the order makes a difference, however. When you're compiling for the MC68020 and use the PC-relative with index mode, the size of the offset to the label depends on whether the label is inside or outside the parentheses. For example, this generates an 8-bit offset to the label:

```
move    @label(d0.w),d0
```

And this generates a 16-bit offset to the label:

```
move    (@label,d0.w),d0
```

Directives

Two or more `DC.B` directives in a row are treated as one `DC.B` directive. This means that a `DC.B` directive can occupy an odd number of bytes when another `DC.B` directive follows. For example:

```
dc.b    'a'
dc.b    'b'
dc.b    'c'
dc.b    'd'
```

is equivalent to:

```
dc.b    'a', 'b', 'c', 'd'
```

However, preceding a `DC.B` directive with a label will word-align it. For example, this:

```
    dc.b    'a'
@1  dc.b    'b'
@2  dc.b    'c'
@3  dc.b    'd'
```

will assemble in the same way as this:

```
dc.b    'a', 0, 'b', 0, 'c', 0, 'd', 0
```

The DC.B directive accepts strings (both Pascal and C) as operands. The terminating null byte is not assembled. Continuing the above example, this:

```
dc.b    'a', 'b', 'c', 'd'
dc.b    5, 'e', 'f', 'g', 'h', 'i'
```

is equivalent to this:

```
dc.b    "abcd"
dc.b    "\pefghi"
```

The DC.W directive can accept a function name or a label as an operand and generate the PC-relative offset of the function or label as a word. The function or label must be defined within the segment in which the directive appears.

Addresses

The difference of two addresses is not a constant expression, so instructions like this are not allowed:

```
move.w  #@2-@1,d0
```

Similarly, the inline assembler does not support the following syntax to assemble the PC-relative offset of the label @1:

```
dc.w    @1-*
```

Instead, use this:

```
dc.w    @1
```

For example, to code a dispatch table, use:

```
; d0 contains 0,1,2,...
    add.w   d0,d0
    add.w   @0(d0.w),d0
    jmp     @0(d0.w)
@0  dc.w    @1          ;  case 0
    dc.w    @2          ;  case 1
    ...                 ;  ...
```

C Calling Conventions

Most of the time, the functions you write will be C functions. They will follow C calling conventions for placing arguments on the stack and returning values in register D0.

C calling sequence

The caller pushes the arguments in right-to-left order, then calls the function. When the function returns, it's the caller's responsibility to remove the arguments from the stack. The caller's code looks something like this:

```
move    ...,-(SP)       ;  last argument
...
move    ...,-(SP)       ;  first argument
jsr     function
add     #...,SP         ;  total size of arguments
```

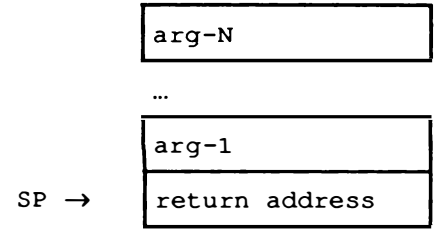
The function's code looks something like this:

```
link        A6,#...     ;  (optional)
...
move        ...,D0      ;  result
unlk        A6          ;  (optional)
rts
```

C function entry

The arguments to the function appear on the stack in right-to-left order. The first (leftmost) argument appears just above the return address, followed by the remaining arguments.

The stack looks like this on entry (just after the call):



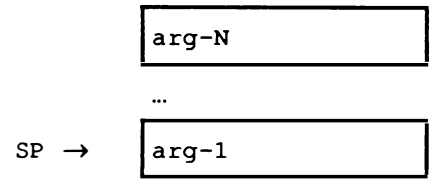
The first argument can be found at 4(SP). If the function begins with a LINK A6,#... instruction, the first argument can also be referenced by 8(A6).

All arguments occupy an even number of bytes on the stack. The caller converts a byte argument into a word. To address this argument as a word use `d(SP)`, where `d` is the appropriate offset. To access the argument as a byte, use `d+1(SP)`.

C function exit

C functions return their result (if any) in register `D0`. The result may be 1, 2, or 4 bytes long. Unused high-order bits may contain garbage.

The stack looks like this on exit (just after the return):



It is the caller's responsibility to remove the arguments from the stack.

Functions that return struct, union, or double

An alternate method is used to return a result of type `struct`, `union`, or `double`, since values of these types are in general too large to fit in `D0`. (Some `structs` or `unions` may be small enough to be returned in `D0`, but the alternate method is used anyway.)

After pushing the arguments, but before issuing the actual call, the caller pushes the address of the location where the return value is to be placed. This address appears at `4(SP)` (or `8(A6)`) and the first argument appears instead at `8(SP)` (or `12(A6)`). The function must get this address and store the result at the location it points to. The address is considered a hidden argument to the function, and it's the caller's responsibility to remove it from the stack.

Because a function returning a `struct`, `union`, or `double` expects its caller to have placed a hidden argument on the stack, it is essential that the caller do so! Therefore, even when you are not interested in the actual return value, always be sure that the function is declared correctly before calling it.

Functions that accept a variable number of arguments

The C calling conventions make it easy to write functions that take a variable number of arguments. The first argument can always be found in the same place regardless of how many additional arguments are supplied. Because responsibility for removing the arguments from the stack lies with the caller, the function doesn't need to clean up the stack.

The standard ANSI library includes the `stdarg.h` header file, which provides a standard interface for all C programs that handle variable arguments. As an elementary example, here is

a function that returns the minimum of an arbitrary number of integers. The first argument is the number of additional arguments passed and must be at least 1.

```
#include <stdarg.h>

int minimum (int count, ...)
{
    va_list xp;                 /* Declare the arg pointer    */
    int x, min;

    va_start (xp, count);       /* Initialize the arg pointer */
    min = va_arg (xp, int);     /* Get the first arg          */
    while (--count) {
        x = va_arg (xp, int);   /* Get the next arg           */
        if (x < min) min = x;
    }
    va_end (xp);                /* Clean up                   */
    return (min);
}
```

> **Note:** You don't need to add the ANSI library to your project to use the `stdarg.h` header file.

A function using Pascal calling conventions cannot accept a variable number of arguments, unless it is written in assembly language.

Pascal Calling Conventions

The Macintosh Toolbox expects that calls to it follow Pascal calling conventions. When you write functions that expect to be called as Pascal functions, be sure to use the `pascal` keyword when you define them. This section tells you how Pascal functions expect to be called.

Pascal calling sequence

The caller pushes the arguments in left-to-right order, then calls the function. Upon return, the result (if any) may be found on the stack. The caller's code looks something like this:

```
clr     -(SP)       ;  reserve space for result
move    …,-(SP)     ;  first argument
…
move    …,-(SP)     ;  last argument
move    function
move    (SP)+,…     ;  result
```

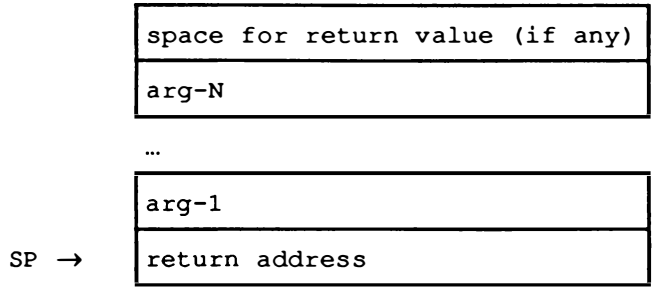
The function's code looks something like this:

```
link    A6,#...         ;  (optional)
...
unlk    A6              ;  (optional)
move    (SP)+,A0        ;  return address
add     #...,SP         ;  total size of arguments
move    ...,(SP)        ;  store return result
jmp     (A0)
```

Pascal function entry

The arguments to the function appear on the stack in left to right order. The last (rightmost) argument appears just above the return address, followed by the remaining arguments in reverse order. If the function returns a result, space for it is reserved above the first argument. If the return value is 1 byte long, 2 bytes are reserved.

The stack looks like this on entry (just after the call):



The last argument can be found at `4(SP)`. If the function begins with a `LINK A6,#...` instruction, the last argument can also be referred to as `8(A6)`.

All arguments occupy 2 or 4 bytes on the stack. A byte argument appears in the high byte of its word and is found at an even offset from `SP` (or `A6`).

Pascal function exit

The function stores its return result (if any) on the stack in the location reserved by the caller. If the result is 1 byte long, it is placed in the high byte of the word reserved.

The stack looks like this on exit (just after the return):

It is the function's responsibility to remove the arguments from the stack.

Tips

Inline assembly can be tricky if you are not familiar with assembly language. It can be especially dangerous if you're used to thinking in terms of high-level languages. The following problems are not specific to THINK C; they are common to assembly language in general.

Using constants

Don't forget the # sign when using an immediate constant. Otherwise, the result will be *very* different than what you intended.

```
extern int MemErr : 0x220; /* declare MemErr low memory global */
    asm {
        move.w  0x220, D0    ;moves contents of location 0x220
                             ;(MemErr) into D0
        move.w  MemErr, D0   ;same thing, but symbolically
        move.w  #0x220, D0   ;moves the value 0x220 into D0
        move.w  5, D0        ;WRONG: On a MC68000 this will cause an
                             ;odd address error. On a MC68020 this
                             ;will move the contents of location 5.
        move.w  #5, D0       ;RIGHT
    }
```

Local storage

Use C variables to declare local storage. If you use the directive DC instead , storage will be allocated in your code segment. **Most of the time, this is not what you want**.

Instruction size

Be sure to use the right-sized instruction when referring to variables. Example:

```
function()
{
    int GetsTrashed;
    int anInt;
    asm{
        move.l  #3, anInt    ;WRONG: will overwrite
                             ;variable GetsTrashed
        move.w  #3, anInt    ;RIGHT: Word size matches int.
    }
}
```

Libraries
13

Introduction

This chapter shows you how to use and build libraries with THINK C. A library is a collection of compiled code you can use in many programs. Libraries usually contain utility functions or interface functions to the operating system. The `MacTraps` library, for instance, contains the interfaces to the Macintosh Toolbox.

Topics covered in this chapter:

- Using libraries
- Using projects as libraries
- Creating libraries
- Converting object files into libraries

Using libraries

To add a library to a project, use the **Add...** command in the **Source** menu. The library name will appear in the project window. When you add a library to a project, its object size is zero. The object code isn't loaded automatically when you add a library.

You can use the **Load Library** command in the **Project** menu to load a library's object code, or you can rely on THINK C's auto-make facility to load it. The auto-make facility will load the library if it was never loaded or if it was changed since it was last loaded.

Creating Libraries

THINK C gives you to ways to create libraries. You can either use any THINK C project as a library, or you can create a separate library document. When you use a project as a library, the smart linker uses only the code it needs. Binary libraries, on the other hand, are smaller, and take up less space on disk.

Projects as libraries

You can use any THINK C project document as a library. When you use a project as a library, THINK C uses smart linking to build your project. The linker links only the CODE components that contain functions that your program uses. (See Chapter 7 to learn about the components of a THINK C project.)

Most of the time, you'll use projects as libraries.

Binary libraries

Use the **Build Library** command in the **Project** menu to create a binary library. When you use this command, you'll see a standard file dialog asking you to name the library.

When you use a library in your project, the linker adds all of the object code in the library to your project.

By convention, libraries end in `.lib`.

Converting object files into libraries

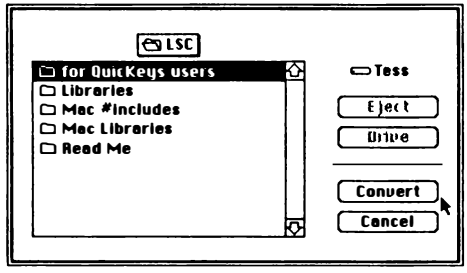
To use object code produced by other compilers and assemblers in THINK C, convert the object files to libraries or projects. Your THINK C package includes two utilities that convert object files.

To convert `.Rel` files produced by Consulair compilers and assemblers into libraries use RelConv. To convert `.o` files produced by Apple's MPW compilers and assemblers into projects use oConv.

Converting .Rel files

The RelConv application converts object files created by Consulair compilers and assemblers into THINK C libraries.

When you double click on the RelConv application, you'll see this standard file dialog:



Double click on the `.Rel` file you want to convert (or click on the Convert button). RelConv generates files with `.lib` suffix. If the name of your object file was `foodle.Rel`, the resulting library will be `foodle.lib`.

When RelConv is finished converting, it will display the standard file dialog again so you can convert more object files. To quite RelConv, select **Quit** from the **File** menu. (You can use the menus even though the standard dialog is active.)

Object files produced by Consulair compilers and assemblers do not retain case information. RelConv assumes that all symbols should be lower case. If you want upper case characters in your symbols, you can supply a vocabulary file.

If RelConv sees a file with the same name as the file it is converting but with a `.voc` suffix, it is considered to be a vocabulary file containing a list of symbols, one per line, with the desired capitalization. Each symbol in the `.rel` file that matches the spelling of a symbol in the vocabulary file will appear with the specified capitalization in the resulting library. Symbols not found in the vocabulary file will appear in lower case.

To help you build a vocabulary file when converting a `.Rel` file for the first time, RelConv, if it finds no vocabulary file, will create one containing all the symbols in the `.rel` file in lower case. You can then edit this file to supply the desired capitalizations and run RelConv again.

RelConv accepts a script file containing a list of `.Rel` files, one per line. The script file must be a text file with an extension of `.rcv`. If relative pathnames appear in the script file, they are interpreted relative to the directory containing the script file.

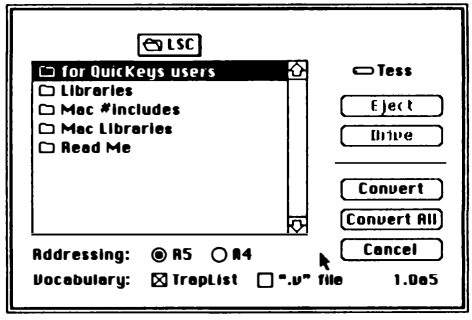
RelConv has an **Options** menu. There is only one option, requesting RelConv to delete each `.Rel` file after converting it.

RelConv will not convert files that were created with the `RESOURCE` directive. This directive is usually used to create resources containing code. THINK C has its own mechanisms for building code resources. See Chapter 7 (You can use the Apple utilities RMaker and ResEdit to create other kinds of resources.)

Converting .o files

The oConv application converts object files created by Apple's MPW compilers and assemblers into THINK C projects. You can use these projects as libraries (see above) or, if you prefer, you can build a library from the resulting project.

When you double click on the oConv icon, you'll see this standard file dialog:



The dialog has some buttons that let you control how the converter builds the project.

The converter displays only files of type 'OBJ ', with creator 'MPS ', and that end in .o. When you double click on a .o file (or click on the Convert button), the converter generates a file ending in .π. For instance, if you convert a file called xyzzy.o, the resulting project file will be called xyzzy.π.

When the converter is through converting the file, it displays the standard file dialog again to let you convert more files. To exit the program, click on the Cancel button.

To convert all the .o files in a folder, click on the Convert All button. The converter doesn't convert files that have already been converted.

The converter lets you choose whether to use A5 or A4 addressing. If you'll be using the project in an application, choose A5. If you'll be using it in a desk accessory, device driver, or code resource, use A4 addressing. To learn more about A4 addressing, read "Global data in drivers" in Chapter 7.

The addressing option only affects those relocatable references where the .o file specifies that PC-relative addressing be automatically substituted as appropriate. Other references to A5 or A4 appearing in the object code cannot be detected; it is the user's responsibility to ensure that they are correct for the kind of program being built. In particular, references to global data may work only with A5-relative addressing.

In some .o files, symbol names appear in all upper case. The vocabulary mechanism provides a way to translate such names back to their proper capitalizations. Only names entirely in upper case will be translated.

When TrapList is checked, the Toolbox and OS glue routines are added to the vocabulary. This includes all the *Inside Macintosh* routines known to THINK C, except those for which THINK C generates inline traps.

When the ".v" file option is checked, the converter examines a user-supplied vocabulary file for each .o file converted. The vocabulary file is a text file that contains the proper capitalization for each symbol, one symbol per line. If the file to be converted is named xyzzy.o, the vocabulary file must be named xyzzy.v.

If the vocabulary file doesn't exist, the converter will create it. The file will contain one line for each symbol that appears entirely in upper case in the .o file. You can edit this file to supply the proper capitalization, and then run oConv again.

THINK C

PART FOUR

Object-Oriented Programming in THINK C

14 Object-Oriented Programming
15 Using Objects in THINK C
16 The THINK Class Library
17–54 The Classes

Object-Oriented Programming 14

Introduction

Object-oriented programming is not hard to learn, but it does require mastering a few key concepts and a few new words. To make learning object-oriented programming easier, this chapter uses examples and comparisons to procedural (traditional) programming. The examples should make some concepts more concrete, and the comparisons to procedural programming will help you relate what you're learning to something you already know.

The basic distinction between procedural and object-oriented programming is in the way the two disciplines treat data and action. In procedural programming, data and action are two separate things. You define your data structures, and then you define some routines to operate on them. For each data structure you define, you need a new set of routines.

In object-oriented programming, action and data are closely coupled. When you define your data— your objects— you also define their actions. Instead of a set of routines that do something to data, you have a set of objects interacting with each other.

Topics covered in this chapter

- Objects and messages
- Classes
- Inheritance and polymorphism
- Objects and the Macintosh interface
- Working with objects
- Where to go next

Objects and Messages

An **object** is an entity that contains some data and an associated set of actions that operate on the data. To request that an object perform one of its actions, you send it a **message**.

For example, you might create an object that represents a rectangle. Its data contains the locations of the rectangle's four corner points, and its actions might include drawing, erasing, and moving. To draw a rectangle, you send the rectangle a draw message.

> **Note:** For the time being, don't worry about *how* you send a message to an object. The important thing is to start thinking of sending messages as a request for an object to do something.

Contrast this with the way you'd do the same thing in procedural programming. First you define a record that represents the four corners of the data structures, and then you define three routines to draw, to erase, and to move a rectangle. Each of the routines would take a rectangle data structure as an argument.

So the first big advantage of object-oriented programming over procedural programming is that you can keep the routines that operate on a data structure together with the data structure they're operating on. This "keeping together" is called **encapsulation**.

Classes

Encapsulation is only a minor advantage of object-oriented programming. A more significant advantage is that objects can inherit data and behavior from other objects.

Every object belongs to a **class**, which defines the implementation of a particular kind of object. A class describes the object's data and the messages it responds to.

Classes are very much like record declarations. You define the private data for the class the same way as you would the fields of a record. In classes, though, the fields are called **instance variables**. Each instance of a class, each object, has its own instance variables just as each variable of a record type has the same fields.

When you send an object a message, it invokes a routine that implements that message. These routines are called **methods**. The class definition includes the method implementations.

One important thing to keep in mind is that message and method are not the same thing. A message is what you send to an object. How an object responds to a message is the method.

You can think of a class as a template for creating objects. It describes an object's data and the messages it responds to. An object is called an **instance** of a class. You can also say that an object is a **member** of a class.

The rest of this chapter uses pictures like this to represent classes. This is what the rectangle class in the example above looks like:

Class Rectangle	
Instance variables top left bottom right	
Messages	**Methods**
Draw	draw line from point to point
Erase	erase lines
Move	offset points

Inheritance and Polymorphism

You can define a class in terms of an existing class. The new class is called the **subclass**, and the existing class is called the **superclass**. A class without a superclass is said to be a **root class**. A subclass inherits all the instance variables and methods of its superclass. Subclasses can define additional instance variables and methods. Subclasses can also **override** methods defined by the superclass.

Overriding a method means that the subclass responds to the same message as its superclass, but it uses its own method to respond to the message.

For example, suppose you want to create a class to represent employees. Two of the instance variables might be the person's name and birth date. You might define three methods, GetName, GetAge and GetWkPay, to return the person's name, to calculate the person's age from the birth date, and to return the person's weekly salary.

Class Employee **Superclass** none	
Instance variables Name Birthdate	
Messages	**Methods**
GetName	return Name
GetAge	return (Today - Birthdate)
GetWkPay	return 0

Now you can use the Employee class to create two new subclasses: HourlyEmployee and ExemptEmployee. Both of these classes inherit all the instance variables from the Employee class. Each class, however, defines a new instance variable to store the salary, and each class overrides the GetWkPay method to return the appropriate weekly salary.

This is what the HourlyEmployee class looks like.

Class HourlyEmployee **Superclass** Employee	
Instance variables HourlySalary	
Messages	**Methods**
GetWkPay	return (HourlySalary * 40)

And this is what the ExemptEmployee class looks like.

Class ExemptEmployee **Superclass** Employee	
Instance variables YearlySalary	
Messages	**Methods**
GetWkPay	return (YearlySalary / 52)

As long as your object is a member of any of the Employee classes, you can send it GetName, GetAge, and GetWkPay messages, and the object will respond properly.

This ability to send the same message to objects of different classes is called **polymorphism**. If you need to create a new kind of employee (a salesperson on commission), all you need to do is define a subclass of Employee and override the GetWkPay method. Any routine that sends GetWkPay messages to employee objects would still work.

Consider for a moment how you would do the same thing in procedural programming. Your employee record would need a flag field to specify whether it was an HourlyEmployee or an exempt employee. Then, your GetWkPay function would check this field and calculate the weekly salary appropriately. If you added a salesperson, you would have to change the *GetWkPay* function to handle the salary calculation for the new kind of employee.

Objects and the Macintosh Interface

The Macintosh interface lends itself to object-oriented programming. For instance, in the Finder, the effect of the **Open** command depends on the icons you've selected. If you select a folder, it opens the folder. If you select an application it launches the application. If you

select a document, it launches the application and tells it to open the selected document. What you're doing is sending an Open message to different Finder objects. Each object uses its own Open method to carry out the request.

Think of the visual entities on the Macintosh screen— windows, controls, the menu bar, icons— as objects. They're things that are waiting to do something. Think of your actions— clicking, dragging, typing— as messages. You're sending messages to the Macintosh objects.

The THINK Class Library, included with your THINK C package, provides a set of classes that let you work with these Macintosh elements from an object-oriented point of view. You can read more about the THINK Class Library in Chapter 16.

Working with Objects

If you've never worked with object-oriented programming, this section will give you some general guidelines for designing classes. As you get more comfortable with object-oriented programming, you'll build up a collection of classes you can use in any of your applications. If you want to see how extensive a library of classes can be, look at the THINK Class Library.

What classes should I define?

Usually, you should define only one root-level class and then define all the other classes as descendants of that class. The advantage of this approach is that you can define behavior that applies to all the objects in your application.

This root-level class is an example of an abstract class. You don't create instances of abstract classes. Instead, you use them to give a family of objects common behaviors. The Employee class in the example above is an abstract class. Its function is to provide the common instance variables and methods you'll need for all employees.

In general, you should define a class (or a subclass of your root class) for each concept your application deals with.

When should I create a subclass?

Create a subclass whenever you need to change the behavior of a method or when you need to add more instance variables. If you write a method that does different things depending on the value of some instance variable, it's probably time to create a subclass.

What should be a method?

If you find yourself passing an object as a parameter to a function to manipulate its instance variables, you should turn the function into a method.

When should I use procedural programming?

Object-oriented programming isn't the answer to all programming problems. Some problems are solved best by "old fashioned" procedural programming. In particular, procedural programming is better for procedures that are highly algorithmic or computationally intensive.

Where to Go Next

The next chapter describes how THINK C implements object-oriented programming. As you'll see, it's a straightforward extension of struct declarations.

Chapter 16 is about the THINK Class Library, a collection of classes you can use to implement Macintosh applications.

Chapters 17–54 describe each class in the `Core Classes` folder of the THINK Class Library in detail. Though these chapters are meant for reference, you should leaf through some of the class descriptions to get a sense of how the THINK Class Library is put together.

Using Objects in THINK C
15

Introduction

This chapter shows you how to use the object-oriented extensions to THINK C. You'll learn how to declare a class, how to declare an object and how to define and call a method.

What you should know

Before you read this chapter, you should know something about object-oriented programming. Particularly, you should know about classes, instances, instance variables, and methods. If you need to learn about these things, look at Chapter 14.

Since THINK C implements objects as handles, you should be familiar with the Macintosh Memory Manager. If you need to learn about the Memory Manager, see *Inside Macintosh II,* Chapter 1, "The Memory Manager."

Topics covered in this chapter

- Overview
- Declaring a class
- Declaring and using objects
- Defining and using methods
- Direct classes
- Tips and techniques
- Summary

Overview

To use objects in your THINK C programs, add the library `oops` (in the `oops Libraries` folder of the `THINK C 1` disk) to your project. If you're building a desk accessory or a code resource, use the library `oopsA4` instead and make sure that you check the Multi-Segment option in the Desk Accessory section or in the Code Resource section of the **Set Project Type...** options dialog.

You'll also need to #include the file `<oops.h>` in the source files that use the functions `new()`, `delete()`, `member()`, and `bless()`. These functions are described throughout this chapter and summarized at the end.

To use objects, you need to declare a class. A class declaration describes the properties of a class and names the instance variables and methods. A class declaration doesn't allocate any space; it merely lets the compiler know what a class looks like.

After you declare a class, you need to define the methods associated with the class. A method is a function that operates on objects of a particular class. A method definition is almost identical to a function definition.

To use objects in a program, you declare a variable, called an **object reference**, that will point to the actual object. Then you use the `new()` function to allocate memory for the object.

Once you've created an object, you can access its instance variables and send it messages. When you're finished using an object, you use the `delete()` function to expunge it from memory.

The sections below talk about each of these steps in detail.

Declaring a Class

A class declaration is like a `struct` declaration. You give the class a name and specify its superclass, then you declare the instance variables and methods like this:

```
struct class : superclass {
   instance variable and method declarations
};
```

The class declaration must specify a class name. If the class has no superclass, it's called a **root class**. To specify that a class is a root class, use either `indirect` or `direct` as its superclass.

> **Note:** The word `indirect` means that objects and subclasses of the class are implemented as handles. The word `direct` means they are implemented as pointers. Unless otherwise specified, the classes mentioned here will be built on `indirect`. For more information on direct classes, see the section "Direct Classes" below.

Following the class name, declare the instance variables and methods. To declare an instance variable, use the same syntax you use to declare a field in a struct. To declare a method, use the same syntax you use to declare a function.

> **Note:** A class declaration must have at least one method declaration.

Here's an example of a class declaration:

```
struct    Window : indirect {
   WindowPtr  theWindow;
   Rect       growRect;
   Rect       dragRect;

   int        Init(void);
   int        Destroy(void);
   int        Hit(Point where);
   long       Drag(Point where);
   int        Draw(void);
};
```

When you declare a class, THINK C automatically `typedefs` the class as if you had typed: `typedef struct class class` so you can use the class name as a type name.

> **Note:** THINK C does not automatically `typedef` structs that aren't classes.

The new class inherits all the instance variables and methods from the superclass. Of course, if the object is a root class, it won't inherit anything. To override a method that's defined in the superclass, just redeclare it in the class declaration.

Your method declarations may contain function prototypes as in the example above. If a method overrides an inherited method, and you provide a prototype, it must match the prototype of the inherited method. If you do not provide a prototype for a method the first time you declare it, no overriding method may contain a prototype.

Declaring circular references

You can create circular references in a class declaration the same way you do for struct declarations. For example, to declare a class with an instance variable of the same class, you would write it like this:

```
struct    AList : indirect {
   int            someInfo;
   struct AList   *nextObj;

   method declarations
};
```

Declaring and Using Objects

Once you've defined a class, you can use its name to declare or define an object reference. An object reference is always declared as a pointer to a value of class type. This is how you declare an object:

Declaring and Using Objects

Once you've defined a class, you can use its name to declare or define an object reference. An object reference is always declared as a pointer to a value of class type. This is how you declare an object:

```
class *object;
```

> **Note:** Objects are always declared as pointers. If you leave out the *, you will get an error "illegal use of class type."

An object reference definition allocates memory for the object reference only. It does not allocate memory for the object. To allocate the memory for the object, you use the `new()` function described later.

Although objects are declared as pointers, THINK C actually implements them as handles. You can use any of the Macintosh Memory Manager calls that work on handles on objects. The section "Objects and the Macintosh" below tells you why you might want to do this.

Two objects of different classes are type-compatible if one class is an ancestor of the other. For assignment, this works in only one direction: an object may be assigned to an ancestor, but not to a descendant.

> **Note:** THINK C uses the assignment rules to match a function call against its prototype.

Creating and deleting objects

Before you can use an object, you must create it to allocate space for it in the heap. When you're done with an object, you'll probably want to delete it to reclaim its space. The functions `new()` and `delete()` create and delete objects.

> **Note:** Be sure to #include `oops.h` in each file that uses the `new()` and `delete()` functions.

To create an object use the `new()` function like this:

```
object = new(class);
```

To delete an existing object, use the function `delete()` like this:

```
delete(object);
```

These two functions are defined in the `oops` library.

> **Note:** Unlike C++, the parentheses surrounding the arguments to `new` and `delete` are required.

Referring to instance variables

To refer to an instance variable of an object, you use the same syntax as when you refer to a member of `struct` through a pointer:

```
object->member
```

Since an object is implemented as a handle, THINK C does one additional indirection for you.

As a rule, only your methods should access an object's instance variables. Your class declaration should provide methods for getting and setting the values of the most important instance variables.

Class membership

THINK C provides two functions to test and force class membership. The `member()` function tells you whether an object belongs to a particular class, and the `bless()` function forces an object to be a member of particular class.

> **Note:** Be sure to `#include oops.h` in each file that uses the `member()` and `bless()` functions.

To test whether an object is a member of a class, use the `member()` function:

```
int member(void *object, void *class);
```

This function returns non-zero if the object's class, or any of its ancestors, is equal to the class specified.

The `bless()` function assigns an object to a particular class:

```
void bless(void *object, void *class);
```

Use this function when you don't use `new()` to create an object. For instance, if you've loaded an object from a resource, use `bless()` to let THINK C treat it as an object of a specific class.

> **Note:** The object you pass to `bless()` must be a handle. Do not use `bless()` to change an object's class arbitrarily.

Defining and Using Methods

You define a method the same way you define a function, except that you include the class name before the method name like this:

```
return-type class::method(parameter declarations)

{
   function body
}
```

Within a method, the compiler supplies an implicit declaration so you can refer to the object that received the message:

```
register class *this;
```

The symbol `this` refers to the object receiving the message. Within a method definition, you can omit `this->` when you refer to an instance variable or when you call a method in the same class as `this`.

If your method definition redeclares a variable with the same name as an instance variable or method, you can use `this->` to recover the original meaning.

For example, for the class defined at the beginning of this chapter, you could define the `Drag()` method like this:

```
long  Window::Drag(where)

   Point   where;

{
   Rect    dragRect;

   dragRect = this->dragRect;
   DragWindow(theWindow, where, &dragRect);
}
```

In this case, `theWindow` refers to the instance variable, and `dragRect` refers to the local variable, while `this->dragRect` refers to the instance variable.

> **Note:** The technique illustrated in this example is useful when you define methods that call Toolbox routines that move memory. Be sure to see section "Objects and the Macintosh" below.

Calling an inherited method within a method

Within a method *definition*, you may want to use the superclass's method. For example, to add functionality to a method, you use the superclass's method, then add more code to augment the behavior.

This is how you call a superclass's method:

```
superclass::method(args)
```

The superclass name must be an ancestor of the current class, and the method must be implemented either in the superclass or one of its ancestors. To refer to the immediate superclass of the current class, you can use the word `inherited`, like this:

```
inherited::method(args)
```

> **Note:** When you call an inherited method, THINK C always bypasses the message dispatcher and generates a direct call.

Calling a method

To call a method (or to send a message to an object), you use this syntax:

```
object->method(args)
```

When you send a messsage to an object, THINK C generates a call to a run-time dispatcher to find out which method it should use. If the method is monomorphic, the linker bypasses the dispatcher and generates a call directly to the method.

> **Note:** A monomorphic method is a method that does not override another method and is never overridden itself.

Direct Classes

Although THINK C uses pointer semantics to refer to the elements of an object, objects are really implemented as handles. Using many handles for many small objects demands a lot from the Macintosh Memory Manager. To avoid overloading the memory manager, or if you want to manage memory yourself, you can use direct classes instead of indirect classes.

If you define the superclass of a root class as `direct` instead of `indirect`, THINK C represents objects of that class with pointers instead of handles. Direct objects are slightly more efficient since they require only one level of indirection to access instance variables. Any subclasses of a direct class inherit the `direct` property.

The `new()` and `delete()` functions are not implemented for direct classes. You're responsible for allocating space for the object yourself.

The `new()` function automatically assigns a class to a newly created object. Since objects of a direct class aren't created with `new()`, you must assign their class before you can use them. To do that, you use the `blessD()` function.

```
void blessD(void *object, void *class);
```

> **Note:** If you forget to allocate space for your direct class object, or if you forget to bless it, your program will almost certainly crash.

Here's an example of creating and blessing a direct class:

```
struct DirObj : direct {
   int         val;
   void    Print();
};

main()
{
   DirObj *myObj;

   myObj = (DirObj *)NewPtr(sizeof(DirObj));
   blessD(myObj, DirObj);

   ... use myObj ...
}
```

If you use a storage allocation function to create a direct object, don't forget that you're also responsible for freeing the space yourself.

Tips and Techniques

This section describes some general tips to make writing object-oriented programs a little easier. It describes how to organize your classes, how to initialize objects easily, how to make sure that your objects behave correctly under the Macintosh Memory Manager, how the sizeof operator works, and how the THINK C implementation of objects relates to C++.

Organizing your classes

It's a good idea to isolate classes to make them easy to use and re-use. For each class, create a `.h` file that contains the class declaration and a corresponding `.c` file that contains the method definitions. The names of the files should be the same as the names of the classes.

Your `.h` file should first `#include` the `.h` file for the superclass of the class you're declaring. Next you declare your class. Since several classes may try to include your header file, you need a way to keep it from being #included more than once. THINK C gives you a way to do this easily.

To prevent an #include file from being included more than once, `#define` a symbol named `_H_FileName`, where `FileName` is the name of the .h file.

Assume you're defining a class called `ClassName`. The file that implements your methods would be called `ClassName.c`. The corresponding header file would be called `ClassName.h`. Here's an example of what `ClassName.h` looks like:

```
#define _H_ClassName  /* To keep from #including it again */

#include "SuperClass.h"

struct ClassName : SuperClass {
   ...
};
```

Initializing objects

Almost every time you create a new object, you'll want to send an `Init()` message to initialize its instance variables. If you set up your `Init()` method to return `this`, you can create and initialize in one step:

```
newTable = ((FixedTable *)new(FixedTable))->Init();
```

The `new()` function returns the object, so the first thing you do — before you assign it to anything — is send it an `Init()` message. Because you've set up `Init()` to return the object, you can assign the fully initialized object to your variable.

Objects and the Macintosh

The most important thing to remember about THINK C objects is that they are implemented as handles. Any reference to an instance variable is implicitly a handle reference. Because of this implementation, your program should not rely on the addresses of instance variables, particularly in calls to Toolbox routines that may move memory.

> **Your program should not rely on the addresses of instance variables, particularly in calls to Toolbox routines that may move memory.**

Look once again at the definition for the `Drag()` method for the Window class declared at the beginning of this chapter:

```
long  Window::Drag(Point where)

{
   Rect    dragRect;

   dragRect = this->dragRect;
   DragWindow(theWindow, where, &dragRect);
}
```

The more obvious definition would be:

```
long  Window::Drag(Point where)

{
   DragWindow(theWindow, where, &dragRect);
}
```

In this case, `&dragRect` refers to the address of an instance variable, in other words, `&this->dragRect`. Since the `DragWindow()` routine may move memory — including the handle `this` — you can't rely on the address of `dragRect` being correct when `DragWindow()` needs it.

The first definition, which copies the instance variable into a local variable with the same name, avoids the problem since the address of a local variables never changes within the function block.

It is OK to assign values to instance variables, even when the expression on the right hand side may move memory. For instance, consider this class declaration:

```
struct FixedTable : indirect {
   Handle theTable;
   int    numItems;

   int    Init();
   ...
};
```

Assume that the `Init()` method allocates a block of 1K bytes like this:

```
FixedTable *FixedTable::Init()
{
   theTable = NewHandle(1024L);
   numItems = 0;
   return this;
}
```

In this case, the assignment to `theTable` is safe. THINK C calls `NewHandle()` before it resolves the reference to `theTable`.

Of course, you can use the Memory Manager routines `HLock()` and `HUnlock()` and `HGetState()` and `HSetState()` to keep the handle `this` from moving around. Most of the time, you won't need to use these routines if you use the "shadowing technique" described above.

The sizeof operator

You can use the `sizeof` operator on class names to get the size of an object. The size of an object is 2 bytes + the sizes of all the instance variables of the object + the sizes of all the instance variables of all its ancestors.

Differences between objects in THINK C and C++

The syntax THINK C uses for objects is loosely based on C++. If you plan on exporting some classes to or from C++, you'll have to keep some things in mind.

THINK C does not implement the keyword `virtual`. You can override any method in a class declaration. To provide compatibility with C++, define a macro to fake the `virtual` keyword:

```
#ifdef THINK_C
#define virtual
#endif
```

To avoid introducing new keywords, THINK C defines `new` and `delete` as functions, so parentheses are always required.

Keywords

No new keywords are reserved. The identifiers `indirect`, `direct`, `this`, and `inherited` are interpreted specially in context. All other identifiers described in this document denote functions.

Summary

This section summarizes the libraries and functions you use to work with objects.

Libraries

If you're building a(n)...	Use this library
application	`oops`
desk accessory	`oopsA4`
device driver	`oopsA4`
code resource	`oopsA4`

Functions

These functions create objects, destroy objects, test membership, and "bless" objects. If you use any of these functions, be sure to `#include <oops.h>` in your source file.

```
void *new(void *class);
```
Create a new instance of the class `class`. You should assign the result to an object reference.

```
void delete(void *object);
```
Delete the specified object. This function releases the memory that the object allocates. It does not release any memory that an instance variable might point to.

```
int member(void *object, void *class);
```
Return non-zero if `object` is a member of `class`. An object is a member of a class if it is of type `class` or if `class` is a superclass of the object's actual class.

```
void bless(void *object, void *class);
```
Treat the memory pointed to by `object` as an object of type `class`. `Object` must be a handle.

```
int memberD(void *object, void *class);
```
Return non-zero if `object` is a member of `class`. An object is a member of a class if it is of type `class` or if `class` is a superclass of the object's actual class. `Object` must be a direct object.

```
void blessD(void *object, void *class);
```
Treat the memory pointed to by `object` as an object of type `class`. `Object` must be a handle. `Object` must be a direct object.

The THINK Class Library
16

Introduction

The THINK Class Library is a collection of classes that implement a standard Macintosh application. If takes care of things like handling menu commands, updating windows, dispatching events, dealing with MultiFinder, maintaining the Clipboard, and so on.

This chapter introduces you to the THINK Class Library and talks about some of the more common topics in application building. After you read this chapter, you should try running the demonstration applications on the disk `THINK C 2`. Use them to explore each of the classes in the THINK Class Library in the chapters that follow this one.

What you should know

This chapter assumes that you're comfortable writing in C. You should be familiar with the fundamentals of Macintosh programming. Particularly, you should know about QuickDraw, the Window Manager, and the Memory Manager.

To use the THINK Class Library, you should know object-oriented programming and how THINK C implements objects. Read Chapter 15 to learn how THINK C implements objects. If you're not familiar with object-oriented programming, Chapter 14 will give you an overview. As you go through this chapter, remember that this is all new territory, so don't be discouraged if it takes a while for all the pieces to fall into place.

Conventions

All the classes in the THINK Class Library begin with the letter 'C' for 'class'. All global variables begin with a lowercase letter 'g'.

Topics covered in this chapter:

- Overview
- Installing the THINK Class Library
- Writing an application with the THINK Class Library
- Working with panes
- Working with menus
- Handling low memory situations
- Undoing and mouse tracking
- THINK Class Library resources
- Modifying the THINK Class Library
- Where to go next

Overview

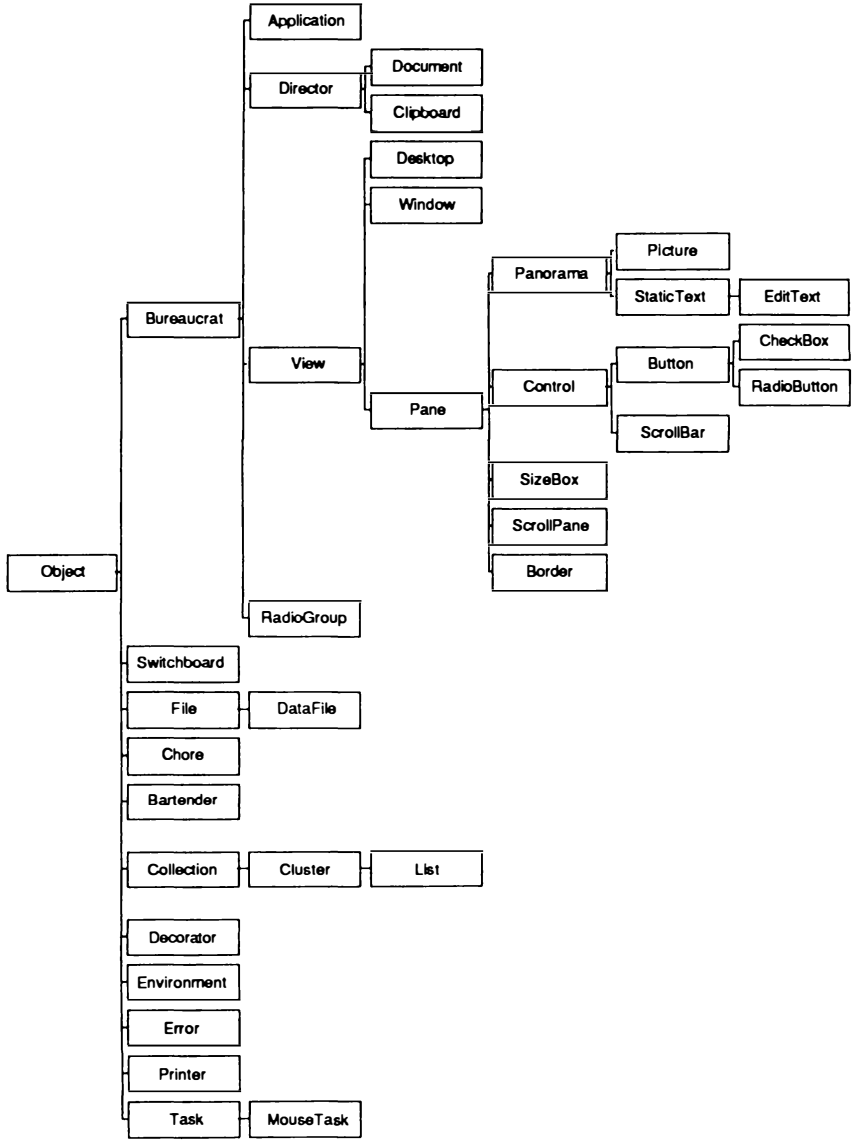
The THINK Class Library is organized into three distinct, interacting structures: the class hierarchy, the visual hierarchy, and the chain of command. The class hierarchy is the set of all the classes that make up the THINK Class Library. The visual hierarchy describes the organization of all visible entities. The chain of command specifies which objects get to handle commands.

The THINK Class Library converts Macintosh events into **visual messages** and **direct commands**. A visual message is an event that affects the visual hierarchy. Mouse clicks, activate events, and update events are all visual messages. A direct command is a request that an object perform an action. Direct commands are usually the result of menu commands.

To convert Macintosh events into messages and commands, the THINK Class Library uses an object called a switchboard. The switchboard calls `GetNextEvent()` or `WaitNextEvent()` repeatedly and, depending on the kind of event, sends a message to the appropriate object. Each application has only one instance of a switchboard object. Another object, called the bartender, takes care of converting menu selections into direct commands.

The class hierarchy

The class hierarchy describes the relationships among all the classes in the THINK Class Library. All of the classes are descendants of the root class CObject. This is what the class hierarchy looks like. To make the drawing easier to read, the initial 'C' of each class is omitted.



Be careful not to think of the class hierarchy as a functional description of the THINK Class Library. You might think from the drawing of the class hierarchy that the CRadioGroup class somehow interacts with the CBureaucrat class, but that's not the case.

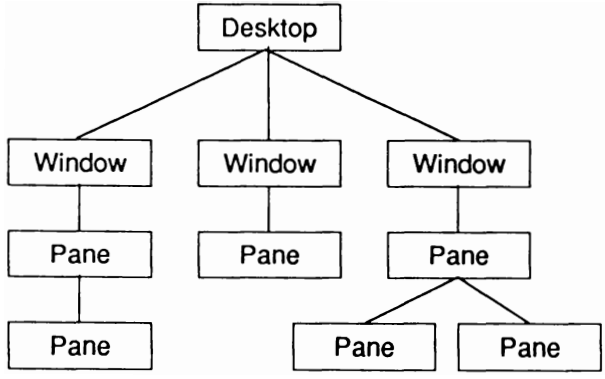
Instead, think of the class hierarchy as a family tree. Each class inherits all of the behavior (*methods*) and all of the attributes (instance variables) of its ancestor. So the CRadioGroup class *is* a bureaucrat. It behaves like a bureaucrat and responds to all of the messages a bureaucrat responds to.

Some of the classes in the class hierarchy are **abstract classes**. The THINK Class Library never creates objects of these classes. They're just used to give a family of objects common behaviors. The most important abstract classes in the THINK Class Library are CObject, CBureaucrat, CView, CDirector, and CApplication.

The visual hierarchy

The visual hierarchy describes all the visible objects, or **views**, that the THINK Class Library knows about. The visual hierarchy is built around the idea of **enclosures**. Everything that you see on the screen belongs to— is enclosed by— another visual entity.

At the top of the visual hierarchy is the **desktop**. The desktop encloses all of the windows in your application. Each window encloses one or more panes. And panes can enclose other panes. This is a typical visual hierarchy.



Unlike the class hierarchy, the visual hierarchy is dynamic. It changes as your program runs. When you open a new document, you add another window to the desktop, and you add another set of panes to the new window.

Panes are the most important kinds of views. All drawing takes place in a pane. The THINK Class Library comes with several different kinds of panes designed for different kinds of displays. Every pane has its own QuickDraw drawing environment, so you can draw in a pane without worrying about where it is on the screen.

Visual events work their way down the visual hierarchy. Since Macintosh update and activate events always have a window associated with them, these messages get sent directly to a window object. Mouse clicks and cursor adjustment messages always work their way down from the desktop to the active window to the appropriate pane.

Panes can handle visual commands like mouse clicks. When you click in a pane, the switchboard determines which pane the mouse went down in and sends it a `DoClick()` message. Because they're descendants of CBureaucrat, you can add panes to the chain of command, so they can respond to direct commands.

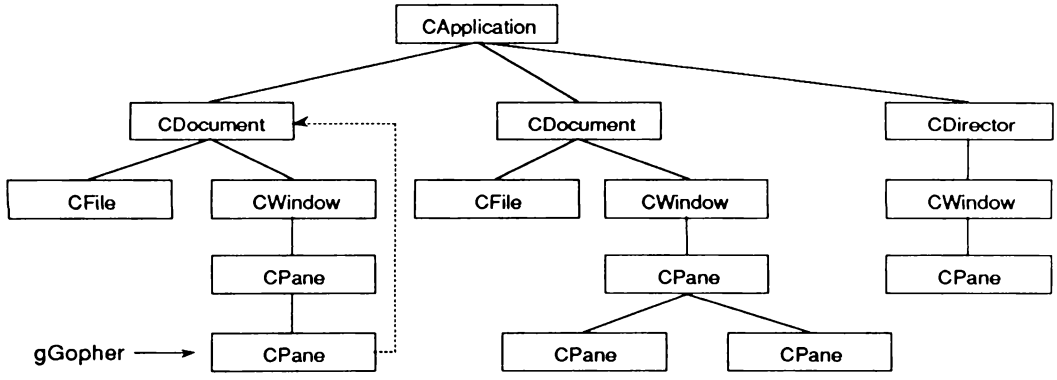
The chain of command

The chain of command specifies which object handles a direct command. The chain of command is based on the idea of **supervisors**. If an object can't handle a direct command, it passes the command on to its supervisor. The application is the only bureaucrat that does not have a supervisor. If the application doesn't handle the command, no object will.

Objects in the chain of command are descendants of the class CBureaucrat. Every bureaucrat has a `DoCommand()` method that the subclass can override to handle specific commands. The default `DoCommand()` method just sends a `DoCommand()` to its supervisor.

The first object to get a chance to handle a command is called the **gopher**. If the object that the gopher points to can't handle the command, it sends the command on to its supervisor. Your application is responsible for setting the global variable `gGopher` to the current gopher.

In this picture, the gopher points to a pane whose supervisor is a document. If the pane can't handle a direct command, it passes the command on to the document. If the document can't handle the command, it gets passed up to the application.



Note that the window and pane which enclose the pane that the gopher points to are not in the chain of command.

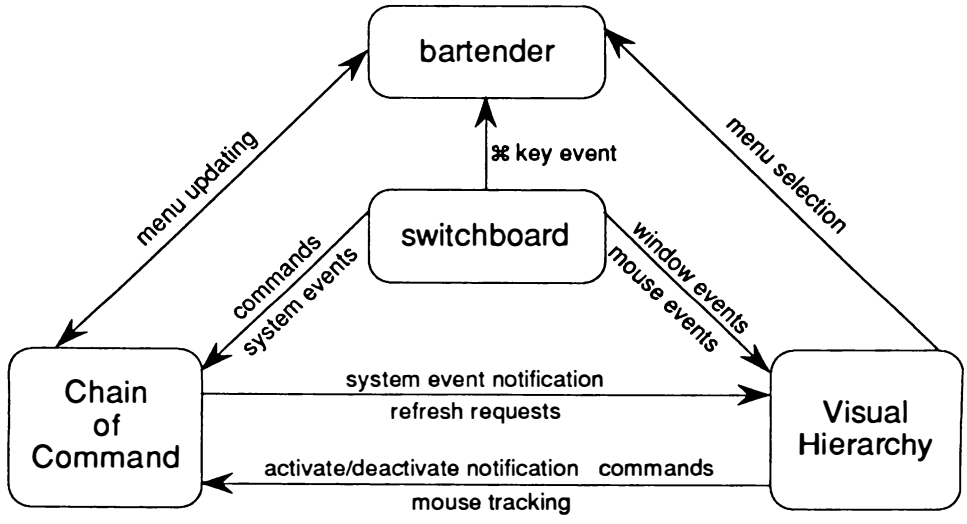
CDirector is an important subclass of CBureaucrat that you need to know about. A director is a bureaucrat that supervises a window. Directors handle the communication between the visual hierarchy and the chain of command. For instance, when a window gets an activate event, it sends an `ActivateWind()` message to its supervisor, which is always a director. The director can then take some action as a result of becoming active.

Another descendant of CBureaucrat is CDocument. A document is a director that has a file associated with it. Documents manage the communication between windows, files, and menu commands. The default document class handles common commands like **Save**, **Save As...**, **Print**, etc. You can think of a document as a file that you view through a window. A better way to think about a document is that it is the essence of a Macintosh application. It is anything that you can display and manipulate inside a window.

The flow of control

The chain of command and the visual hierarchy get their direction from the switchboard. The switchboard gets events from the Macintosh Event Manager and converts the event into messages for either the chain of command or the visual hierarchy. Messages for the chain of command usually go to the gopher, the first bureaucrat in the chain. Messages for the visual hierarchy go to the active window or to the desktop.

This diagram shows you where different kinds of events go after the switchboard converts them to messages:



When you press the mouse, the switchboard sends a `DispatchClick()` message to the desktop. If the click was in the menu bar, the desktop sends the bartender a message to update the state of the menus before they appear. The bartender sends a message to the chain of command to enable and disable the appropriate menu items. Then the desktop uses the bartender to convert the menu selection into a direct command and sends a `DoCommand()` message to the gopher.

If the click was in a window, the desktop sends a `DispatchClick()` message to the window, which eventually sends a `DoClick()` message to the pane the mouse went down in. The pane's `DoClick()` method can then do whatever's appropriate for the pane. It might even send a `DoCommand()` message to an object in the chain of command.

When the switchboard gets an activate or an update event, it sends an `Activate()`, `Deactivate()`, or `Update()` message to the window. The window sends a similar message to its director.

When you type, the switchboard sends a `DoKeyDown()` or a `DoAutoKey()` message to the gopher. If you hold down the Command key when you type, the switchboard asks the bartender to convert the key into a command and sends a `DoCommand()` message to the gopher.

If you're running under MultiFinder and you bring another application to the foreground, the switchboard sends a `Suspend()` message to the application (not the gopher) which sends `Suspend()` messages to all of its directors. A similar thing happens when your application comes to the foreground.

> **Note:** The THINK Class Library treats desk accessories as if they were in their own layer, even if you're not using MultiFinder, so your application will still get suspend and resume "events" when you bring up a desk accessory.

Installing the THINK Class Library

If you want to use the THINK Class Library, it's important that you install it correctly on your disk. First, be sure that you've followed the instructions in Chapter 2 to install THINK C on your disk.

> **Note:** The files that make up the core of the THINK Class Library take up nearly 500K of disk space. While it may be possible to use the THINK Class Library on a floppy-based system, it is not recommended.

Installing the Class Library

Copy the folder `THINK Class Library` from the disk `THINK C 3` into your `THINK C Folder`. This is what's in the `THINK Class Library` folder:

File or Folder	Description
Core Classes	This folder contains two folders. Core Sources contains all the .c files that comprise the core of the class library. Core Headers contains all the header files for Core Sources. Chapters 17–54 describe every class in this folder.
More Classes	This folder contains additional classes. More Sources contains the .c files, and More Headers contains the header files. The demonstration program Art Class on the disk THINK C 3 uses both the Core Classes and the More Classes. (The Art Class sources are on disk THINK C 4). This manual doesn't describe the classes in the More Classes folder.
TCL Resources	This resource file contains all the standard resources you need to use the THINK Class Library. For more information about this file, see "Resources" later in this chapter.
TCL TMPLs	This file contains a set of TMPL resources that make it easy to create resources for pane intialization. The next set of instructions tell you how to use these TMPLs.

Installing the TMPL resources into ResEdit

If you use ResEdit to create and edit your resource files, you should install the TMPL resources from TCL TMPLs into your copy of ResEdit.

> **Note:** ResEdit is included in your package on the disk THINK C 4.

The TMPL resources are resource editor templates for ResEdit. After you install them, it will be easy for you to create certain THINK Class Library resources with ResEdit. Follow these steps to install the TMPL resources into ResEdit.

- Make a duplicate copy of ResEdit.
- Open the duplicate copy of ResEdit you just made
- Open the file TCL TMPLs
- Select the item TMPL
- Choose **Copy** from the **Edit** menu
- Open the original copy of ResEdit
- Choose **Paste** from the **Edit** menu
- Choose **Quit** from the **File** menu
  When ResEdit asks you to confirm the changes, click on the Yes button.
- Delete your copied version of ResEdit

Installing the demonstration programs

Your THINK C package includes four demonstration program to show you how to use the THINK Class Library. Three of these are fairly simple and use only the core classes of the class library. The other program is more advanced and shows off some of the things you can do with the THINK Class Library.

The three simple demo programs are in the disk `THINK C 2` in the folder `TCL Demos`.

Folder name	Description
`Starter Folder`	This project just opens and closes empty windows. It's a minimal project you should use as the basis of your own projects.
`Pedestal Folder`	Another "do nothing" project. This one is a little different from the Starter project and more extensively commented.
`TinyEdit Folder`	A small text editor based on the Starter project.

Copy these folders from the disk `THINK C 2` to to your `Development` folder. You should not put these folders in your `THINK C Folder`.

Try running the demonstration programs. The first time you run one of these programs, it will need to compile all of the THINK Class Library files, so it may take a while, particularly if you're using a Macintosh Plus. After the first time, compile times will be much faster since you won't need to recompile any of the class library files.

Writing an Application with the THINK Class Library

This section tells you how to write an application with the THINK Class Library. The easiest way to do this, is to take the Starter demo program and build from it.

To create an application with the THINK Class Library, you just need to create subclasses of existing classes. The three classes you need to override are CApplication, CDocument, and CPane. Your application subclass determines the overall structure of your application. The document subclass implements the way your application handles its files, and the pane subclass implements how the information in your file appears in the document windows.

In addition to the subclasses, you also need a resource file for your project. This resource file must contain the standard THINK Class Library resources as well as your own. The standard THINK Class Library resources are in the file `TCL Resources`. When you use the THINK Class Library, make a copy of this resource file and name it the same as your project and append `.rsrc` to it. Then add your own resources to it.

> **Note:** For more information about resource files and the THINK Class Library, see "Resources" below. To learn about using resources with a project, see "Using resource files with projects" under "Anatomy of a Project" in Chapter 7.

Creating the project

The best way to begin a project is to use the Starter application (in the `Starter Folder` of the disk `THINK C 2`).

- Copy the Starter Folder and change its name to the name of your application
- Open the new folder
- Rename Starter.π to the name of your application
- Rename Starter.π.rsrc to the name of your project plus .rsrc
- Open the project
- Use the **Find...** command in the **Search** menu to change Starter to the name of your application
- Use the **Save As...** command to save the files under your own names (the **Save As...** command also changes the names of the files in the project)

Use the **Set Project Type...** dialog to make sure that your project is Multi-Finder Aware and will receive Suspend & Resume events.

> **Note:** Be sure you set these options, even if you're not running with MultiFinder. Otherwise, your application will behave strangely when you do run it under MultiFinder.

Creating the application subclass

If your application is a standard Macintosh application, your application subclass must override these methods:

```
SetUpFileParameters          CreateDocument
OpenDocument                 DoCommand
```

The `SetUpFileParameters()` method sets up the standard file parameters that specify which files are visible in the standard file box when the user chooses **Open...** from the **File** menu.

The `CreateDocument()` method creates a new, untitled document. The document it creates is one that you define as a subclass of `CDocument`. After creating the document, your `CreateDocument()` method should send it a `NewFile()` message. This is the method that gets called when the user chooses **New** from the File menu.

The `OpenDocument()` method is like the `CreateDocument()` method. Instead of sending the newly created document a `NewFile()` message, though, this document should

send it an `OpenFile()` message. The `OpenDocument()` method takes one parameter, an `SFReply` record, that contains the information about the file that the user chose to open.

The `DoCommand()` method handles all the application-specific commands. Most of the commands should be handled at the document level. Some commands, like **New**, **Open**, and **Quit**, are handled by the default application class.

Creating the document subclass

The document is where your application draws and displays its data. All documents have windows associated with them. Most documents also have an associated file. Neither the window nor the file are created automatically. You must create them yourself.

Your document class should override these methods:

```
Initialization method          OpenFile()
Dispose()                      DoSave()
DoCommand()                    DoSaveAs()
NewFile()                      Revert()
```

Your document class must have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By convention, the name of your initialization method should be `IYourDoc()` where `YourDoc` is the name of your document class. Your initialization method should call `CDocument::IDocument()`. The supervisor of a document is always `gApplication`.

If your application allocates memory, you should also override the `Dispose()` method to deallocate it. Be sure that your method calls `inherited::Dispose()` to make sure that the document is disposed of properly.

> **Note:** You do not need to dispose of the `itsWindow` or the `itsFile` instance variables. The default `Dispose()` method does that for you.

Your document class's `DoCommand()` method does most of the work in your application. When a window is active, the switchboard will send all commands to the document first (it's the gopher), and if the document can't handle it, the application class tries to handle it. Your document class should handle all the commands it knows about, and call `inherited::DoCommand()` when it can't.

> **Note:** Be sure that your `DoCommand()` method or that one of the methods it invokes sets the instance variable `dirty` to TRUE when there has been a change to the document.

Your document class will get a `NewFile()` message when the user chooses **New** from the **File** menu. This method needs to create a window and attach the panes for it. The `NewFile()` method doesn't need to create a file until the user tries to save the document.

Your document gets an `OpenFile()` message when the user chooses **Open...** from the **File** menu. The `OpenFile()` method has one argument: a pointer to a Macintosh `SFReply` record. When you get the `OpenFile()` message, you can be sure that the `SFReply` record is properly filled in. Your `OpenFile()` message needs to create an instance of a file object (usually of class CDataFile). You can send your file any of several read messages to obtain its contents. Your `OpenFile()` method also needs to create a window to display the contents of the file, just as your `NewFile()` method does.

When the user chooses **Save** from the **File** menu, your document gets a `DoSave()` message. Your `DoSave()` method should write the contents of its file to disk. The file object is stored in the instance variable `itsFile`.

When the user chooses **Save As...** from the **File** menu, your document gets a `DoSaveAs()` message. This method takes an `SFReply` record, and you can be sure that it is properly filled in. Your document class needs to override this method to write its data to a file.

If your application supports the **Revert** command, you should implement it in the `DoRevert()` method. Your implementation might do the same thing as closing without saving, then opening the file again.

Creating the pane subclass

Once you've created your windows and opened your files, you need to display them somewhere. In the THINK Class Library, you don't write directly onto the window. Instead, you create a subclass of class pane.

When you create a subclass of Pane, you need to override these methods:

```
IYourPane                    Dispose
Draw                         DoClick
```

Your pane class should have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By convention, the name of your initialization method should be `IYourPane()` where `YourPane` is the name of your pane class. Your initialization method should call `CPane::IPane()` (or whatever pane class you're overriding). The supervisor of a pane should be either the pane that encloses it or the director its window belongs to.

The pane initialization method is where you set the pane's location in its enclosure and its characteristics. If you want your pane to receive clicks, be sure to send the pane a `SetWantsClicks(TRUE)` message, otherwise mouse clicks in your pane are ignored.

If your pane allocates memory, you should also override the `Dispose()` method to deallocate it. Be sure that your method calls `inherited::Dispose()` to make sure that the pane is disposed of properly.

The `Draw()` message tells your pane to draw its contents. You can assume that the port, clip region, and coordinate system have been set up correctly. See "Drawing in the THINK Class Library" in the next section to learn all about drawing in a pane.

When you click in a pane, it will get a `DoClick()` message. Your `DoClick()` method can handle the mouse click itself to draw something, to drag an object, or to select something. Some panes, like the edit text pane implemented by CEditText, have built-in `DoClick()` methods.

If you want your mouse action to be undoable, you need to create a subclass of CMouseTask and send it in a `TrackMouse()` message. The section "Mouse Tracking" below goes into more detail about undoable mouse actions.

Working with Panes

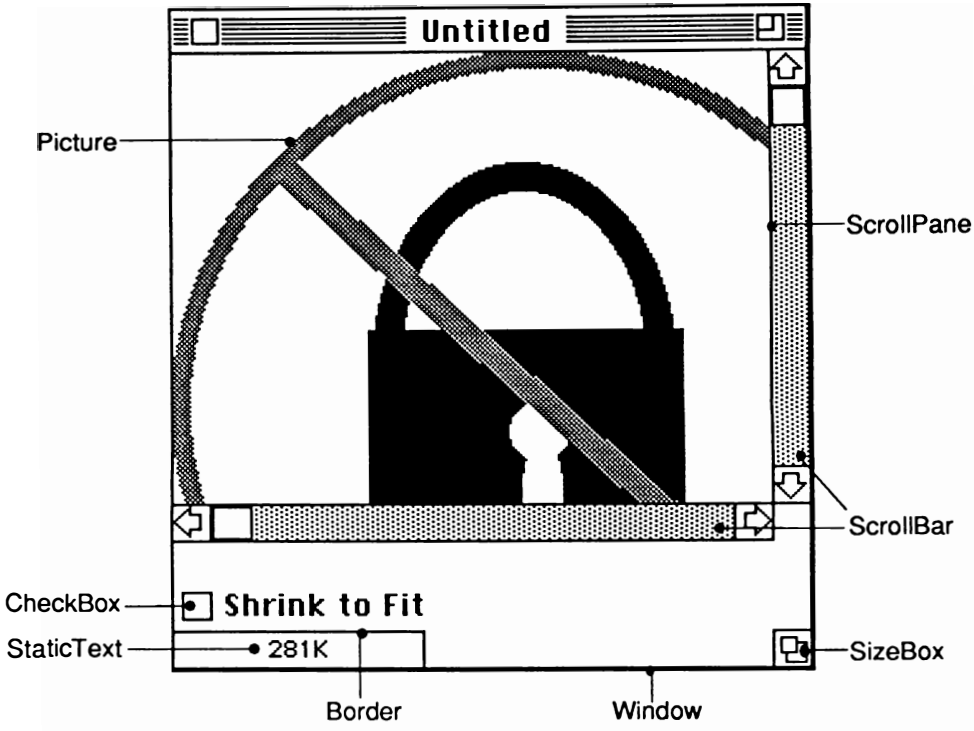
In the THINK Class Library, all your drawing takes place in a pane. A pane is a rectangular region of the screen completely enclosed by a window. A window may have several panes, and each pane may have several subpanes. Each pane has its own drawing environment and can handle its own visual commands.

Every pane has an enclosure that completely encloses the pane. A pane also has a supervisor— an object in the chain of command. A pane's enclosure and supervisor can be the same object, particularly if the pane belongs to another pane. In most the common case, though, the supervisor is the director that owns the window or the pane.

Windows and panes

All panes are subclasses of the abstract class CView which defines the behavior of visual entities. A window is a view, but it is not a pane. Other visual entities are subclasses of panes; they include things like controls, borders, pictures, and size boxes.

Every pane belongs to a window or to another pane. The outer, or owning pane, is the enclosure. Every pane has one enclosure. For example, in this picture of a window, there are eight panes (the window is not a pane):



The size box, the check box, the border, and the scroll pane are enclosed by the window. The two scroll bars belong to the scroll pane. The picture is enclosed by the window, and the static text belongs to the border.

Drawing in a pane

Every pane has its own drawing environment. The rectangle that describes the edges of a pane is the pane's **frame**. The frame defines a QuickDraw drawing environment for the pane. In most cases, the top, left corner of the pane is the point (0, 0).

To draw in a pane, you override its `Draw()` method. The THINK Class Library sends your pane a `Draw()` message whenever the pane needs to be updated. You can send a `Refresh()` message yourself if you want to force an update event.

To draw in a pane use the standard QuickDraw routines. The pane's `Prepare()` method sets up the QuickDraw port and coordinate system, so you don't have to worry about where

the pane is. Your pane gets a `Prepare()` message before it gets a `Draw()` or a `DoClick()` message, so the coordinate system is always set up correctly.

You can draw directly in a pane as a result of a mouse click. In this case, you need to override the `DoClick()` method of the pane. If you want to make your mouse action undoable, use a mouse task. See the section "Undoing and Mouse Tracking" below.

Properties of panes

When a pane moves or changes size, all of the panes that it encloses change as well. How a pane changes depends on its **sizing characteristics**. When you create a pane, you specify its horizontal and vertical sizing characteristics.

The horizontal sizing characteristic specifies how the pane's left and right edges change.

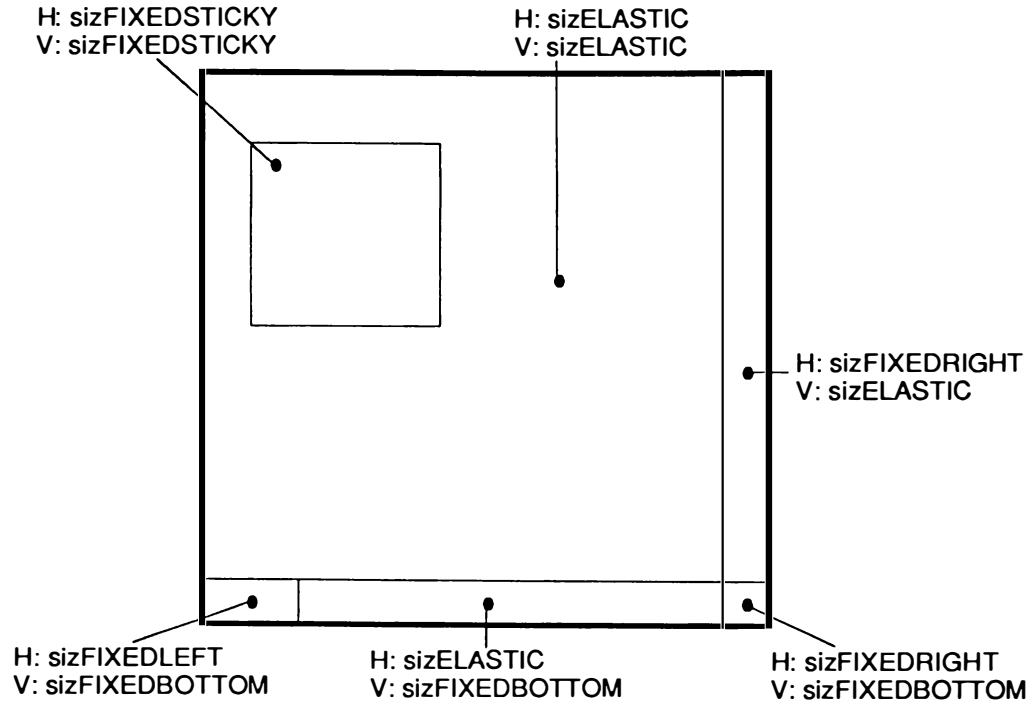
Horizontal sizing	Meaning
`sizFIXEDLEFT`	The left edge of the pane is always the same number of pixels from the left edge of the enclosing pane as when it was originally placed.
`sizFIXEDRIGHT`	The right edge of the pane is always the same number of pixels from the right edge of the enclosing pane as when it was originally placed.
`sizFIXEDSTICKY`	The left and right edges stick to their original locations in the enclosing pane. If the enclosure scrolls horizontally, the pane will scroll with it.
`sizELASTIC`	The width of the pane grows or shrinks by the same amount as the width of the enclosing pane.

The vertical sizing characteristic specifies how the pane's top and bottom edges change.

Vertical sizing	Meaning
`sizFIXEDTOP`	The top edge of the pane is always the same number of pixels from the top edge of the enclosing pane as when it was originally placed.
`sizFIXEDBOTTOM`	The bottom edge of the pane is always the same number of pixels from the bottom edge of the enclosing pane as when it was originally placed.
`sizFIXEDSTICKY`	The top and bottom edges stick to their original locations in the enclosing pane. If the enclosure scrolls vertically, the pane will scroll with it.
`sizELASTIC`	The height of the pane grows or shrinks by the same amount as the height of the enclosing pane.

A couple of examples might help: A vertical scroll bar in a window has the characteristics `sizFIXEDRIGHT` and `sizELASTIC`. It has a fixed horizontal length and remains anchored to the right edge of the window. Vertically, it changes with the height of the window. A status box in the lower left corner of a window would be `sizFIXEDLEFT` horizontally and `sizFIXEDBOTTOM` vertically. It has a constant size and remains anchored to the bottom left corner of the window.

In this picture, the dark line represents the window. It contains a main pane that takes up most of the window and several other panes. The two panes that hold the scroll bars are fixed to the edges of the window. When the window is resized, they grow or shrink by the same amount as the window. The panes that hold the grow box and the status box are anchored to the bottom corners of the window. The square pane in the upper left portion of the main pane will always be there, regardless of how the window changes. Its dimensions will not change automatically.



Coordinate systems

When you're working with panes in the THINK Class Library, you need to know about three coordinate systems. It may seem like a lot, but you're already familiar with two of the THINK Class Library's coordinate systems.

The desktop and some internal methods and Toolbox routines use **global coordinates**. In this coordinate system, all units are in pixels, and (0, 0) is at the top, left corner of the main screen.

To position a pane within a window, you use **window coordinates**. In this system, the top, left corner of the window's content region is (0, 0). In this system, too, each unit is one pixel. Aside from setting the position of a pane within a window, you'll almost never need to use window coordinates.

> **Note:** Window coordinates are useful if you want to give two different panes a common point of reference.

Frame coordinates provide a QuickDraw coordinate system for your pane. Units in frame coordinates are in pixels, and the point (0, 0) is usually the top left corner of the pane. If the pane moves within its enclosure, the coordinate system does not change; the top left corner is still (0, 0). The only time this origin point changes is when you scroll the pane. You can use the Toolbox call `LocalToGlobal()` to convert frame coordinates into global coordinates.

The CPane class has several methods that transform frame coordinates to window coordinates and vice versa. It also defines methods to convert pane coordinates into the coordinates of its enclosure.

Panoramas

Almost everything you want to display is bigger than a pane. Graphics and text, for instance, frequently take up more room than what you can fit in a pane. The THINK Class Library provides a **panorama** class, CPanorama, that lets you display portions of a large graphic in a pane. You might say a panorama is a pane that scrolls.

Think of a panorama as a sheet of paper glued to a desk. The frame moves over the paper. The only part of the panorama you can see is what's inside the frame. To scroll, you move the frame around on the panorama to see different parts of it.

The rectangle that completely encloses the panorama is called the **bounds rectangle**. The bounds rectangle defines the size and coordinate system of the panorama. Usually, the top, left corner of the bounds rectangle is the point (0, 0), and the units in its coordinate system are pixels.

The coordinate system of the bounds rectangle specifies how the frame moves over the panorama when you scroll. In the usual case, when you scroll up, you move the pane up a pixel. In some applications, though, you want to scroll more than one pixel. In a text editing application or in a spreadsheet application, you want to move up by an entire line.

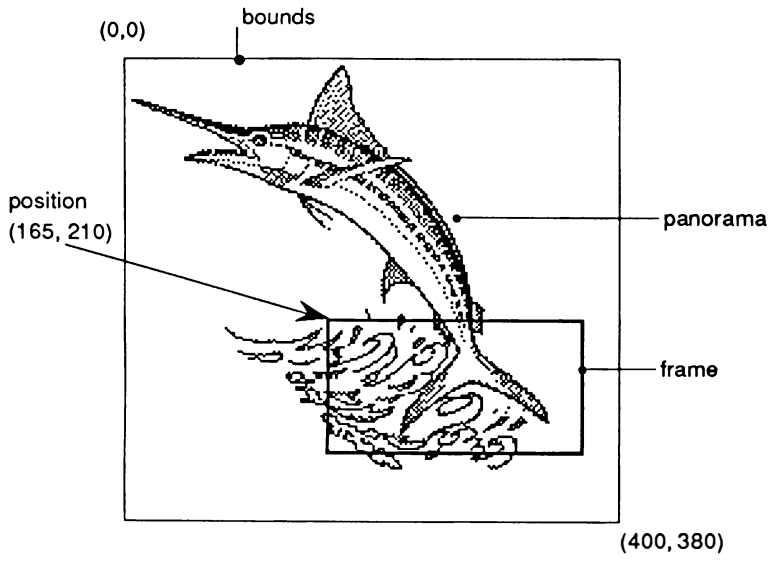
You can specify a **scale** that says how many pixels make up a single panorama unit. You can set different scales for horizontal and for vertical units. In a graphics application, each unit might be one pixel. In a spreadsheet application, a vertical unit might be 12 pixels, and a horizontal unit might be 60 pixels.

The units of the panorama bounds are for scrolling only. For drawing, you'll use the frame coordinates which are always pixel units.

There are two ways to talk about the top, left corner of the frame of the pane. The top, left corner of the pane, expressed in panorama units, is the **position** of the frame in the panorama. The top, left corner, expressed in frame coordinates, is the **origin** of the frame.

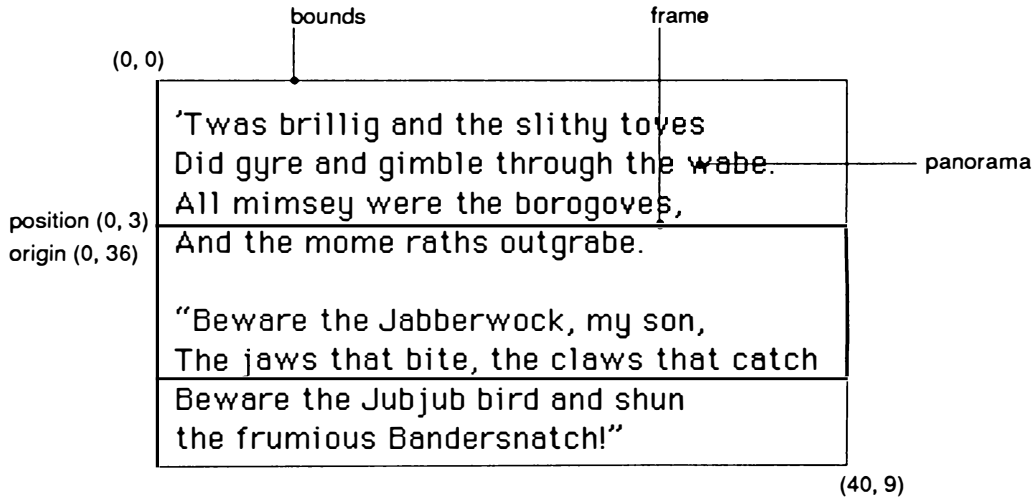
The key thing to remember is that scrolling always happens in panorama units. Drawing always takes place in frame coordinates. As you scroll, the origin of the frame changes. A couple of examples might help.

In this picture, both the horizontal and vertical scales are set to 1 pixel per panorama unit. The bounds rectangle of the panorama is (0, 0, 400, 380). The portion of the picture you can see in the frame of the pane, the fish's tail, starts at (165, 210) in the panorama.



Since the panorama units match the frame units, the position of the frame in the panorama and the origin of the frame are the same. If you were to draw a line from (220, 230) to (270,230), it would cut across the tail of the fish.

In the following picture of some text, the horizontal scale is 6 pixels per unit, and the vertical scale is 12 pixels per unit. The bounds rectangle of the panorama is (0, 0, 40, 9). In the panorama scale, this means 8 lines of 40 characters each. The position of the frame in the panorama is (0, 3), the beginning of the fourth line. The origin of the frame, though, is at (0, 36). If you wanted to draw a line to strike out the word "And," you would draw it from (0, 42) to (18, 42)



Note: The top, left of the bounds rectangle doesn't have to be (0, 0). You can define the bounds coordinate system that's most convenient for the kind of data you're displaying in the panorama.

Scroll panes

To make it easy to use panoramas, the THINK Class Library provides a class called CScrollPane that implements a **scroll pane**. Scroll panes give you an easy way to attach scroll bars to your panorama.

You create a scroll pane the same way as any other pane. You can request a vertical scroll bar, an horizontal scroll bar, and a size box. Then you use the `InstallPanorama()` method to associate a panorama with the scroll pane. The scroll pane examines the panorama and adjusts the scroll bars appropriately. You can specify how many panorama units to scroll when you click on different parts of the scroll bar.

The scroll bars and the panorama communicate through the scroll pane. When you click in one of the scroll bars, it tells the scroll pane which tells the panorama how much to scroll.

Cursor tracking

The `AdjustCursor()` method that all panes inherit from CView lets you change the cursor when it moves into your pane. Most of the time you'll use only one cursor in the whole pane. In this case, all you have to do is set the cursor with the Toolbox routine `SetCursor()`. Look at the `AdjustCursor()` method in CEditText for an example.

Sometimes, though, you might want to use different cursors within the same pane. The `AdjustCursor()` method lets you do this as well, but it takes a little more work. See the description of the CView class in Chapter 52.

Initializing views from resources

The `IViewRes()` method lets you initialize any descendant of CView from a resource template.

Resource	Class
`Bord`	CBorder
`Pane`	CPane
`Pano`	CPanorama
`PctP`	CPicture
`ScPn`	CScrollPane
`StTx`	CStaticText, CEditText
`View`	CView

You can use ResEdit to create the resource templates for each class.

> **Note:** Your THINK C package includes a file `TCL TMPLs` that contains `TMPL` resources you can install into ResEdit. These `TMPLs` let you create and edit the resources above. See the instructions in "Installing the THINK Class Library" above to learn how to install these `TMPLs` into ResEdit.

When you use ResEdit to create view resource templates, keep in mind these values for sizing and clipping mnemonics.

Sizing values

```
sizFIXEDLEFT = 0
sizFIXEDRIGHT = 1
sizFIXEDTOP = 2
sizFIXEDBOTTOM = 3
sizFIXEDSTICKY = 4
sizELASTIC = 5
```

Clipping values

```
clipAPERTURE = 0
clipFRAME = 1
clipPAGE = 2
```

Working with Menus

The THINK Class Library lets you think of your menu commands more abstractly than the Macintosh Toolbox. Instead of identifying a menu command by its menu ID and item number, the THINK Class Library lets you assign unique **command numbers** to each item of the menu. With command numbers, you can reassign functions to different menu items without having to rebuild the application.

Command numbers are positive long integers in the range 1 to 2,147,483,647. Command numbers in the range 1 to 1023 are reserved for the THINK Class Library. Command number 0 is reserved for `cmdNull`, the null command. All other command numbers (1024 to 2,147,483,647) are available for your application.

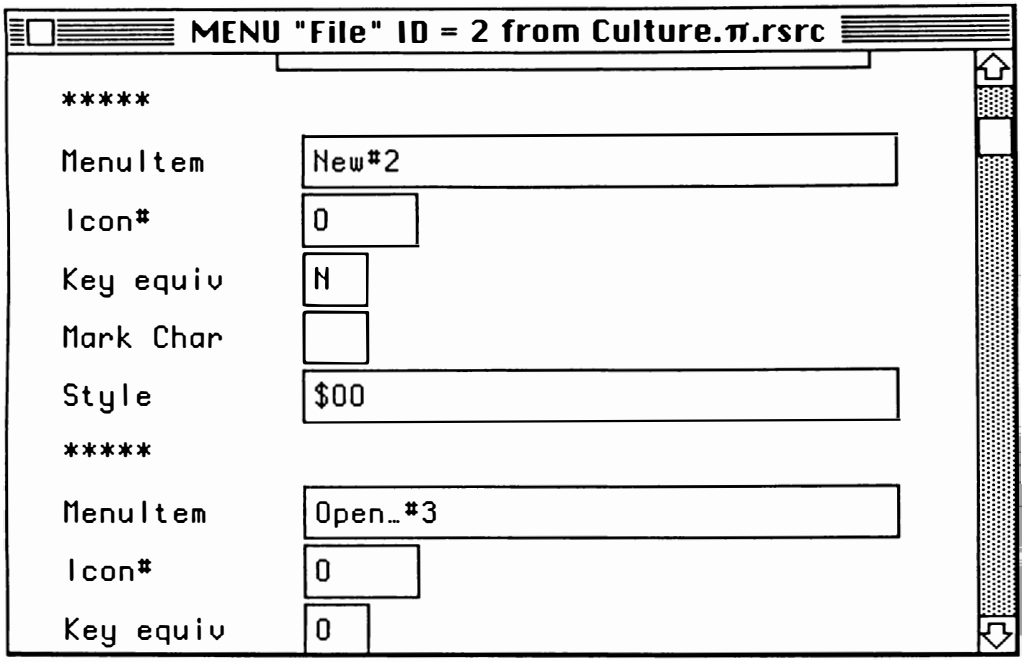
The reserved commands are for the most common Macintosh application commands like **Open**, **Save**, **Quit**, **Page Setup...**, etc. The reserved commands are in the file `Commands.h`. Be sure you use the reserved command number if you expect to get the default behavior from the THINK Class Library.

When you choose a command from the menu bar, the desktop sends a `FindCmdNumber()` message to the bartender. The bartender matches the menu ID and the item number to a command number and sends the gopher the command number in a `DoCommand()` message. If the you use a Command key equivalent, the switchboard sends the `DoCommand()` message to the gopher.

> **Note:** Remember, the gopher is a pointer to a command object. Usually, the gopher points to a pane within the active window.

Using MENU resources

When you create your menus with ResEdit, append the command number to the menu item. The menu item and the command number are separated by the character #. For example, the **File** menu looks like this:



Note: If you don't add append a command number to a menu item, the bartender automatically assigns it `cmdNull`.

You can use virtually any menu ID for your application's menus. The THINK Class Library reserves the following menu IDs for certain menus:

Menu title	Menu ID	Mnemonic
Apple	1	`MENUapple`
File	2	`MENUfile`
Edit	3	`MENUedit`
Font	10	`MENUfont`
Size	11	`MENUsize`

After you create all the menus that your application needs, create an `MBAR 1` resource that contains all of the IDs of the menus your application uses. The application's `SetUpMenus()` message creates the global bartender (stored in the global variable

gBartender) to read the MBAR 1 resource. The bartender creates the tables that match command numbers to menu items.

> **Note:** Your application's menus **must** be in MBAR 1 unless you change the defnition of MBARapp in Constants.h.

If you want the bartender to return the menu ID and item number of a particular menu item, use the special command number -1 in your MENU resource. The bartender will return the negative of the menu ID in the high word and the menu item number in the low word. This is the same as menus that you build on the fly, described next.

Building menus on the fly

Menus that you create as your program is running, like Font menus, won't have command numbers associated with them. In this case, the bartender's FindCmdNumber() method responds by returning a negative number. The high word of this number is the negative value of the menu ID. The low word is the item number. When your DoCommand() method gets a negative command number, you know you have to figure out the command from the menu ID and item number.

For example, if DoCommand() gets a command -655351 (0xFFF60009), it means that there is no command number associated with the menu item. The high word, 0xFFF6, is -10, and the low word, 0x0009, is 9, so the user chose the 9th item of the menu with ID 10.

You can add menu items to existing menus if you like. For example, you might want to add the Font menu to a general text-handling menu, or you might want to have a menu with the names of all the documents your application has opened. The important thing to remember is to add all these menus at the **end** of the existing menu. Otherwise, the bartender will get confused.

Dimming and checking menu items

The bartender includes methods to let you enable and disable and check and uncheck menu items. When you click in the menu bar, the bartender sends an UpdateMenus() message to all of the bureaucrats in the chain of command. In the general case, all the items in the menu start out dimmed and unchecked. Then each bureaucrat enables the menu items that pertain to it. Once the appropriate items have been enabled and checked, the Toolbox routine MenuSelect() displays all the menus.

> **Note:** The dimming, undimming, and unchecking take very little time. You won't notice a delay between the time you click on the menu bar and when the menu is displayed.

Suppose you click in the menu bar of a text processing application. When you click on the menu bar, but before the Toolbox displays the menu, the bartender disables all the menu items. Then, the application enables all the application related menu items: **New**, **Open...**,

Quit, for example. The document enables all the document related items: **Save**, **Save As...**, **Revert** (if the document's been changed), and so on. A pane might check the current font and size in the **Font** menu. Finally the menu appears on the screen with the correct items checked and enabled.

The `UpdateMenus()` method of your application, document, and pane need to enable each item. To make sure that item enabling happens from the general (application) to the specific (pane), be sure to call `inherited::UpdateMenus()` first in your own `UpdateMenus()` method.

You can use the bureaucrat methods `SetDimOption()` and `SetUnchecking()` in your application `SetUpMenus()` method to modify this behavior. `SetDimOption()` lets you specify whether the bartender should dim all, some, or none of the items when you click on the menu bar. For Font menus, for instance, it doesn't make sense to dim all the font names only to reenable them again.

Dim Option	Meaning
`dimNONE`	Never dim any of the menu items.
`dimSOME`	Dim only the menu items that have command numbers associated with them.
`dimALL`	Dim all of the menu items. Each bureaucrat's `UpdateMenus()` method must enable the items for the commands it handles. This is the default.

There are only two options for `SetUnchecking()`. The bartender either unchecks all the items or it doesn't.

Unchecking option	Meaning
`TRUE`	Uncheck all the menu items at menu selection. Your `UpdateMenus()` method should check the appropriate items. Set this option for menus like **Font** menus or **Style** menus.
`FALSE`	Don't uncheck any menu items at menu selection. This is the default since most menu items never need to be checked.

Using pop-up menus

If you want to use pop-up menus in your application, you can use the regular Macintosh Toolbox commands and deal with the menu as you would without the THINK Class Library. If you want to register your menus with the bartender, send it an `AddMenu()` or an `InsertInBar()` message. After you call the Toolbox routine `PopUpMenuSelect()`,

pass the high word and low word of its return value in a `FindCmdNumber()` message to the bartender.

Handling Low Memory Situations

The CApplication class has several methods to deal with low memory situations. These methods involve the rainy day fund and credit limit values you specify in the application initialization method. This section gives a quick overview. For more detail, see Chapter 17.

If the Macintosh Memory Manager gets a request for more memory than is available, it calls a grow zone function to get more memory. In the THINK Class Library, the grow zone function sends the application a `GrowMemory()` message.

The `GrowMemory()` method tries several strategies to free memory in the heap. First, it sends the application a `MemoryShortage()` message. Your application subclass should override this method to release memory that is not crucial to execution. If the `MemoryShortage()` method wasn't able to release enough memory, `GrowMemory()` starts using the application's memory reserves.

When you initialize your application, you tell it how many bytes to set aside in a **rainy day fund**. This rainy day fund is the application's memory reserves. The way `GrowMemory()` uses the rainy day fund depends on the **credit limit** you specify in the application's initialization method and on the values of the `canFail` and `loanApproved` flags.

If the number of bytes needed to satisfy a memory request is less than the credit limit, `GrowMemory()` takes the memory from the rainy day fund. If the `loanApproved` flag is set, `GrowMemory()` tries to get as much of the memory from the rainy day fund as it can, depleting it entirely if necessary.

If there still isn't enough memory to fulfill the request, `GrowMemory()` looks at the value of the `canFail` flag. If the flag is set, `GrowMemory()` simply returns, and it is up to your application to deal with the failed request. If the flag is not set, `GrowMemory()` sends the application an `OutOfMemory()` message. The default `OutOfMemory()` method posts an out-of-memory error message and jumps to the beginning of the event loop. You may want to override this method to quit the application gracefully or to make a last ditch effort to release memory.

Undoing and Mouse Tracking

The THINK Class Library provides a class that lets you implement the **Undo** command easily. In the default implementation, each document has its own undo history.

Undoing

The THINK Class Library uses the abstract class CTask to implement undoable actions. For every undoable action you want in your application, you need to create a subclass of CTask.

After you perform an action, you store enough information to undo it in your task's instance variables. Then you send the task to your supervisor in a `Notify()` command. The CDocument class implements the `Notify()` method to store a task in one of the document's instance variables. When you choose **Undo** from the **Edit** menu, the document's `DoCommand()` method sends an `Undo()` message to the task it stored.

Here's an example. Suppose you've defined a subclass of CTask to change the font in an edit text pane. Before passing the command on to the edit pane's `DoCommand()` method, you create a task and store the current font in an instance variable. After you pass the font command to the edit text pane, you send the task in a `Notify()` message to the document.

Your `Undo()` method would simply send the font change command to the document. Since the command goes through the regular command chain, your `DoCommand()` method would create a task to let you undo what you were undoing.

Mouse tracking

The THINK Class Library uses the undo mechanism to make mouse tracking easier and undoable. The CMouseTask class is an abstract class that defines some methods specifically for mouse tracking.

To implement a mouse tracking task, define a subclass of CMouseTask and override the `KeepTracking()` and `EndTracking()` methods. The `KeepTracking()` method does whatever you want to happen while the mouse is down. The `EndTracking()` method does whatever you want to happen when the mouse is released.

For example, if you're moving a rectangle from one place in a pane to another, the `KeepTracking()` method might draw a gray outline that moves as you move the mouse. The `EndTracking()` method would erase the rectangle from its old location and redraw it in the new location.

If you want to make your mouse task undoable, you need to store enough information in the object to undo the effects of mouse tracking. You must also override the `Undo()` method *(inherited from* CTask) to use this information to undo the effects of the mouse task.

After you've tracked the mouse, you can send the task in a `Notify()` message to the document. When you choose **Undo** from the **Edit** menu, the document sends an `Undo()` message to the task to undo the effects of mouse tracking.

THINK Class Library Resources

The THINK Class Library requires certain resources in your project's resource file. All of the resources described in this section are in the file `TCL Resources` on the disk `THINK C 3`. The mnemonic constants for all of these resources are in the file `Constants.h` in the `Core Headers` folder.

Alerts

The `ALRT` and `DITL` resources always have matching IDs. You can change these resources to suit your application.

ALRT/DITL	Used for
128	General. A handy, all-purpose alert box. The `DITL` contains only `^0`, so you can use the Toolbox routine `ParamText()` to set up the text.
150	Confirm to revert to last saved version.
151	Confirm to save changes before closing or quitting.
200	Severe Macintosh error has occurred.
250	No printer selected.
300	Macintosh OS error alert.

Controls

The THINK Class Library uses this `CNTL` template for all the scroll bars it creates.

CNTL	Used for
300	Scroll bar

Error message strings

The THINK Class Library uses a resource of type `Estr` to report Macintosh errors. `Estr` resources have exactly the same format as `STR` resources. You can use the ResEdit command **Open as Template...** in the **File** menu to open and edit an `Estr` resource as a `STR`.

The ID of the `Estr` resource is the error code you want to identify. The file `TCL Resource` includes one `Estr`. The error handling class CError uses `Estr` resources to display messages. You should create an `Estr` for every error your application reports to the user.

Estr	Used for
-192	Tried to get nonexistent resource

Menus

The THINK Class Library reserves these menu IDs for the standard menus. The **File** and **Edit** menus contain all the standard items. You can remove the ones that don't apply to your

application. The bartender builds the desk accessory menu for you automatically, but you'll have to build the **Font** and **Size** menus yourself in the `SetUpMenus()` method of your application. For an example of Font and Size menus, see the `TinyEdit` project in the `TCL Demos` folder of disk `THINK C 2`.

MENU	Used for
1	Apple
2	File
3	Edit
10	Font
11	Size

> **Note:** The mnemonics for these menus are in `Commands.h`, not in `Constants.h`.

Menu bars

The THINK Class Library uses this resource to install all the menus in your application. The `MBAR` resource in `TCL Resources` automatically includes the **Apple**, **File**, and **Edit** menus.

MBAR	Used for
1	List of all menus to install at application startup.

Small Icon

The THINK Class Library uses this small icon to draw a grow box instead of the Toolbox routine `DrawGrowIcon()`. If your application uses `SICN`s, you can use the routine `DrawSICN()` in the file `TBUtilities.c` to draw it in a pane.

SICN	Used for
200	Grow box

Strings and string lists

The THINK Class library uses these strings for various prompts and messages. You can modify these to suit your application.

STR	Used for
150	Prompt for the **Save As...** dialog box.
300	Generic operating system error message used when no `Estr` resource is available.

STR#	Used for
128	List of common Macintosh words. This list includes the words: quitting, closing, Undo, Redo, and Untitled

129	Strings used for low memory warnings.
130	Task names for changing the wording of the **Undo** menu item text. This resource has no strings. Your application should add strings to this list if it supports Undo. See the descriptions of CTask and CMouseTask.

Window template

The THINK Class Library requires only one window template for the Clipboard window. Your application will probably define one or more additional `WIND` templates.

WIND	**Used for**
200	Clipboard. Window template used for displaying the clipboard.

Modifying the THINK Class Library

You can modify the THINK Class Library classes to suit your particular needs. To change the behavior of one of the THINK Class Library classes, create a new subclass of the class you want to modify, and add new methods or override the methods you need to change. In some cases, it might be a better idea to change the source code for a class. If you need to keep track of events as the Event Manager gets them, it's probably easier to modify the CSwitchboard class to do this than to create a subclass.

Be aware that there are dangers in changing the source code of a class. From time to time, Symantec may release new versions of the THINK Class Library. Changes *that you* make in the source of a class may make it difficult to use new classes or updates of existing classes. If you do make changes to the source of a class, be sure to keep an archival copy of the original class and to mark your changes clearly.

The second danger is a little more subtle. As you use the THINK Class Library, you may want to create new classes and subclasses to implement some kind of behavior. If you want to be able to use these classes in other programs, and especially if you want others to be able to use these classes, you should not rely on any features of your modified classes.

> **Note:** Under your license agreement, you may distribute new subclasses of the classes in the THINK Class Library. You may not, however, distribute modified sources of the classes in the THINK Class Library.

Where to Go Next

Learning to use the THINK Class Library takes time and experimentation. Start with the `TinyEdit` example in the `TinyEdit Folder` of `THINK C 2`. It was built from the application you should use to create your own applications. You might start by adding an

About... box to the `TinyEdit` application. (Hint: Since it's an application-wide command, implement it in the application's `DoCommand()` method.) As you explore how the `TinyEdit` application was put together, look at the next chapters to understand how the classes of the THINK Class Library work.

If you want to see what's possible with the THINK Class Library, look at the Art Class demonstration program. The finished application is on the disk `THINK C 3`. The sources for it are in the `Art Class Folder` on disk `THINK C 4`.

CApplication 17

Introduction

Every program must create a subclass of CApplication and create only one instance of this class. The application is the highest level in the chain of command. It is the only bureaucrat without a supervisor.

Heritage

Superclass	CBureaucrat
Subclasses	You must define a subclass of this class.

Using CApplication

CApplication is one of the classes you must override to implement an application with the THINK Class Library. An application usually has one or more subordinate **directors** which handle the interaction between commands and windows. The most common kind of director is a **document** which, in addition to a window, also handles a file.

The application and the chain of command

The application is the root of the chain of command. If the current command object, or **bureaucrat**, (stored in the global variable `gGopher`) can't handle a command , it passes the command to its supervisor . If no bureaucrat can handle the message, it ends up with the application. If the application doesn't handle a command, the message is ignored.

Writing the main program

Your main program must create an application object and assign it to the global variable `gApplication`. After initializing your application, you need to send your application a `Run()` message to get it started. Finally, you send it an `Exit()` message to give it a chance to clean up after itself. Your `main()` routine should look like this:

```
void main()
{
   gApplication = new(CYourApp);
   ((CYourApp*)gApplication)->IYourApp();
   gApplication->Run();
   gApplication->Exit();
}
```

Note: In this example, CYourApp is the name of your application subclass.

Your application subclass must override these five methods:

```
IYourApp()                      OpenDocument()
SetUpFileParameters()           DoCommand()
CreateDocument()
```

Of course, your subclass can override additional methods if it needs to.

Handling low memory situations

The CApplication class provides several methods that deal with low memory situations. These methods involve the rainy day fund and credit limit values you specify in the application initialization method. Before talking about what values to use for these parameters, it helps to know what happens when you run out of memory.

If the Macintosh Memory Manager gets a request for more memory than is available, it calls a grow zone function. In the THINK Class Library, the grow zone function sends the application a `GrowMemory()` message.

The `GrowMemory()` method tries several strategies to free memory in the heap. First, it sends the application a `MemoryShortage()` message. Your application subclass should override this method to release memory that is not crucial to execution. For instance, your `MemoryShortage()` method might release a code segment it's not using, or it might dispose of a buffer it no longer needs. If the `MemoryShortage()` method wasn't able to release enough memory, `GrowMemory()` starts using the application's memory reserves.

When you initialize your application, you tell it how many bytes to set aside in a **rainy day fund**. This rainy day fund is the application's memory reserves. The way `GrowMemory()` uses the rainy day fund depends on the **credit limit** you specify in the application's initialization method and on the values of the `canFail` and `loanApproved` flags.

> **Note:** Use the `RequestMemory()` message to set the `canFail` and `loanApproved` flags. Never set them directly.

If the number of bytes needed to satisfy a memory request is less than the credit limit, `GrowMemory()` takes the memory from the rainy day fund. If the `loanApproved` flag is set, `GrowMemory()` tries to get as much of the memory from the rainy day fund as it can, depleting it entirely if necessary.

> **Note:** When the rainy day fund is used and not replenished, the application displays a warning message at the next idle event.

If there still isn't enough memory to fulfill the request, `GrowMemory()` looks at the value of the `canFail` flag. If the flag is set, `GrowMemory()` simply returns, and it is up to your ap-

plication to deal with the failed request. If the flag is not set, `GrowMemory()` sends the application an `OutOfMemory()` message. The default `OutOfMemory()` method posts an out-of-memory error message and jumps to the beginning of the event loop. You may want to override this method to quit the application gracefully or to make a last ditch effort to release memory.

The value you use for the rainy day fund should be slightly larger than the amount of memory your application allocates and releases in one pass through the event loop. The amount you specify for the credit limit should be enough to accommodate normal Toolbox operations like region manipulation. You might want to try 20K for the rainy day fund and 5K for the credit limit.

Use the `RequestMemory()` method to set the values of the `loanApproved` and `canFail` flags. For instance, when you want to allocate memory for a document and there isn't enough memory, you can just display a "too many documents open" message. This is how you do it:

```
    /* Set the loanApproved flag to false so the */
    /* operation doesn't tap the rainy day fund  */
    /* Set the canFail flag to indicate that     */
    /* it's OK if the request fails              */
gApplication->RequestMemory(FALSE, TRUE);

    /* Make the request */
myHandle = NewHandle(50000L);

    /* Set the flags back to normal */
gApplication->RequestMemory(FALSE, FALSE);

    /* Now it's your responsibility to make sure */
    /* that the memory was actually allocated    */

if (myHandle == NULL) {
    /* There wasn't enough memory for that */
    /* Tell the user to close some documents */
}
else {
    /* The request was successful. Go on */
}
```

The key things to remember are to surround your large memory requests with `RequestMemory()` messages and to set the flags to FALSE after your memory request.

Variables

The global variable `gApplication` points to your application object. Your main program needs to set this variable.

The instance variables of the application class handle events, memory shortage situations, standard file parameters, and flow of control. Your application subclass may define additional instance variables.

Global variable

`CApplication *gApplication;`	The single instance of your application object.

Event related instance variables

The event related instance variables handle the interaction between the Macintosh Toolbox Event Manager and your application. Your application subclass should not manipulate any of these variables.

`CSwitchboard *itsSwitchboard;`	Points to the event-processing object.
`CCluster *itsDirectors;`	Supervisors for windows and documents.
`CList *itsIdleChores;`	Chores to perform at idle time.
`CCluster *itsUrgentChores;`	Chores to perform after the current event.
`Boolean urgentsToDo;`	Are any urgent chores pending?
`Boolean running;`	If TRUE, the program is still running.

Memory related instance variables

The memory related variables are used when your application is running out of memory. The `rainyDayFund` variable specifies the number of bytes to set aside to use when your application runs into a critical memory situation. You specify this amount in your application initialization method.

`Size rainyDayFund;`	Bytes of memory to set aside for critical memory situations.
`Handle rainyDay;`	Handle to the reserve memory.
`Boolean rainyDayUsed;`	Has rainy day fund been tapped?
`Boolean memWarningIssued;`	Has the user been alerted?
`Size creditLimit;`	Maximum number of bytes allocated without approval.
`Boolean loanApproved;`	Is the memory request approved?
`Boolean canFail;`	Is it OK for memory request to fail?

Standard file instance variables

The standard file variables are used in the `ChooseFile()` method as parameters to the Macintosh Toolbox `SFPGetFile()` routine. You set these variables in your `SetUpFileParameters()` method.

`short sfNumTypes;`	Number of file types recognized
`SFTypeList sfFileTypes;`	File types which are recognized
`ProcPtr sfFileFilter;`	Filter for files to display
`ProcPtr sfGetDLOGHook;`	Hook for handling get dialog
`short sfGetDLOGid;`	Dialog resource ID for get file Default is `getDlgID` (-4000). It's #defined in `StdFilePkg.h`
`ProcPtr sfGetDLOGFilter;`	Filter for get dialog events

Flow of control instance variable

This variable is used to return the flow of control to the beginning of the event loop.

`JumpBuffer eventLoopJump;`	Saved stack frame to jump to on errors.

Methods

The methods of the CApplication class deal with application initialization and document handling. Since the application is the ultimate supervisor for all the command objects, it also serves as the end of the chain of command.

Initialization methods

The initialization methods set up the application's memory and file parameters. Your application subclass should implement its own initialization method and call the default initialization method.

```
void IApplication(short extraMasters, Size aRainyDayFund, Size
aCreditLimit);
```

This is the main initialization method. In your application subclass, you should implement a method called `IMyAppClass`, where `MyAppClass` is the name of your application subclass.

The `extraMasters`, `aRainyDayFund`, and `aCreditLimit` parameters are passed to the `InitMemory()` method described later on.

Your application subclass must call `CApplication::IApplication()` in its initialization method. If your application subclass defines additional instance variables, your initialization method should initialize the variables.

IApplication() initializes the following instance variables. To learn what these variables are for, see the section "Instance variables" above.

```
itsSwitchboard                  itsIdleChores
itsDirectors                    itsUrgentChores
```

This method also initializes these global variables. All the global variables are described in detail in Chapter 54.

```
gHasWNE                         gWatchCursor
gSleepTime                      gGopher
gError                          gLastViewHit
gIBeamCursor                    gClicks
gUtilRgn                        gSignature
```

Finally, this method sends these messages to your application object. The rest of this section goes into detail for each method.

```
InitToolbox()                   MakeDecorator()
InitMemory()                    SetUpFileParameters()
MakeDesktop()                   SetUpMenus()
MakeClipboard()
```

```
void InitToolbox(void);
```

This method initializes all of the Macintosh Toolbox managers. Your application subclass should not need to override this method.

```
void InitMemory(short extraMasters, Size aRainyDayFund, Size
aCreditLimit);
```

This method initializes the memory manager and sets up the GrowZone() function used in low memory situations. ExtraMasters is the number of times to call the Toolbox routine MoreMasters(). ARainyDayFund is the number of bytes to set aside in the application heap to deal with low memory situations. ACreditLimit is the largest number of bytes that can be used from the rainy day fund without approval.

```
void MakeDesktop(void);
```

This method creates the desktop and stores it in the global variable gDesktop. The default method creates a standard desktop. If your application uses a non-standard desktop, one that supports floating windows, for instance, you should override this method in your application subclass.

> **Note:** The class CFWDesktop in the More Classes folder implements a floating window desktop.

```
void MakeClipboard(void);
```

This method creates the clipboard and stores it in the global variable `gClipboard`. If you create your own subclass of `CClipboard`, you should override this method to create a clipboard of your class.

```
void MakeDecorator(void);
```

This method creates the window decorator and stores it in the global variable `gDecorator`. The window decorator is responsible for the default sizes of windows and arranges them neatly on the desktop. If you want to implement a different decorator, you can create a subclass of `CDecorator` and override this method to use your decorator.

```
void SetUpFileParameters(void);
```

This method lets you set up the parameters that specify what kinds of files your application works on. The THINK Class Library passes some of these parameters to the Toolbox `SFPGetFile()` function which displays the standard get file dialog. Your application subclass should override this method and call `inherited::SetUpFileParameters()` to set up the default value.

Your own method should then set the following instance variables and globals:

`sfNumTypes`	The number of different types of files your application deals with. If your application can open any kind of file, set this value to -1.
`sfFileTypes`	The file types of the files that your application deals with. One type for each element of this array.
`gSignature`	The four character signature of your application. Although it's not an instance variable, this method is the place to set up this global variable.
`sfFileFilter`	Optional. A pointer to a function that filters the file names your application can deal with.
`sfGetDLOGHook`	Optional. A pointer to a dialog hook function for the standard get file dialog.
`sfGetDLOGid`	Optional. The resource ID of the standard get file dialog you're using. Do not change this variable if you want to use the default dialog
`sfGetDLOGFilter`	Optional. A pointer to a dialog filter proc

To learn about file filter functions, dialog hook functions, and dialog filters, see *Inside Macintosh I*, Chapter 20, "The Standard File Package."

```
void SetUpMenus(void);
```

This method creates the bartender object, which handles all interactions with the menu bar and menu items, and stores it in the global variable `gBartender`. `SetUpMenus()` reads all

the menu information from your application's MBAR 1 resource and builds the menus. It also builds the desk accessory menu.

If your application uses menus that need to be built on the fly, you should override this method. For instance, if your application uses a Font menu, you would build it in this method. Be sure to call `inherited::SetUpMenus()` to make sure all the regular menus get built. See the description of CBartender for more information about building these kinds of menus.

Command methods

The command methods handle events that the application takes care of. Since messages frequently get passed up the chain of command— from the pane, to the document, to the application— the application is the last chance to handle command messages. Most of the command methods don't do anything. They're null methods that keep the message from being passed on.

```
void Notify(CTask *theTask);
```

When a subordinate object finishes a task, it sends the task to its supervisor. Documents usually handle `Notify()` messages. The default `Notify()` method for applications doesn't do anything. It's here in case a document tries to pass the `Notify()` message on to the application. Your application subclass should not override this method.

```
void DoKeyDown(char theChar, Byte keyCode, EventRecord *macEvent);
```

You should handle key-down events within a pane or a document. For applications, this method doesn't do anything. It's here in case the `DoKeyDown()` message gets passed up the chain of command. Your application subclass should not override this method.

```
void DoAutoKey(char theChar, Byte keyCode, EventRecord *macEvent);
```

You should handle auto-key events within a pane or a document. For applications, this method doesn't do anything. It's here in case the `DoAutoKey()` message gets passed up the chain of command. Your application subclass should not override this method.

```
void DoKeyUp(char theChar, Byte keyCode, EventRecord *macEvent);
```

You should handle key-up events within a pane or a document. For applications, this method doesn't do anything. It's here in case the `DoKeyUp()` message gets passed up the chain of command. Your application subclass should not override this method.

> **Note:** The Macintosh system event mask is usually set to mask out key-up events. If you need to get key-up events, be sure to reset the system event mask with the Toolbox routine `SetEventMask()`.

```
void DoCommand(long theCommand);
```

This is the only command method that really does anything. `DoCommand()` handles application commands that the user chooses from the menu. These are the commands that the default `DoCommand()` method handles:

Command	Action
`cmdNew`	Sends a `CreateDocument()` message to the application.
`cmdOpen`	Sends a `ChooseFile()` message to your application. If the reply is good, sends an `OpenDocument()` message to your application. The default `ChooseFile()` message calls `SFPGetFile()` with the values you specified in the `SetUpFileParameters()` method. Your application must override the `OpenDocument()` method.
`cmdClose`	If the front window is a desk accessory, close it. Your document class should handle this command to close documents.
`cmdQuit`	Sends a `Quit()` message to your application. The default `Quit()` method sends a `Quit()` message to each of your application's documents. The default `Quit()` method for documents closes the document. Any document can cancel quitting.

`cmdUndo`
`cmdCut`
`cmdCopy`
`cmdPaste`
`cmdClear`
	If the front window is a desk accessory, always call `SystemEdit()`. Your document class should handle these commands to edit documents.

`cmdToggleClip`	Sends a `Toggle()` message to the clipboard.

If your application defines its own application-related events, your application class should override this method. If your `DoCommand()` method gets a command that it does not handle, it should call `inherited::DoCommand()`.

```
void UpdateMenus(void)
```

This method enables the appropriate menu items right before a menu selection. The default method enables the **Quit** command. If the application is in the foreground, it enables the **Show/Hide Clipboard** command. If the application is in the background, this method enables the `cmdClose`, `cmdUndo`, `cmdCut`, `cmdCopy`, `cmdPaste`, and `cmdClear` commands for desk accessories.

Your document's `UpdateMenus()` method should enable the appropriate Edit menu commands for the document.

Memory management methods

These methods deal with critical memory situations in your application. The `InitMemory()` method sets the function `GrowZoneFunc()` (defined in `CError.c`) as the function to call in low-memory situations. This function sends a `GrowMemory()` message to your application to try to reclaim enough memory to continue.

```
void RequestMemory(Boolean aLoanApproved, Boolean aCanFail);
```

If `aLoanApproved` is TRUE, subsequent memory requests can use the rainy day fund to satisfy memory requests. If `aCanFail` is TRUE, it's up to the application to handle the case where memory could not be allocated. You should send this message to the application before and after all large memory requests. Here's an example:

```
gApplication->RequestMemory(FALSE, TRUE);
myHandle = NewHandle(50000L);
gApplication->RequestMemory(FALSE, FALSE);
if (myHandle == NULL) {
   /* couldn't get 50000 bytes */
   /* do something else (alert the user ) */
}
else {
   /* go on with this operation */
}
```

The first `RequestMemory()` says not to use the rainy day fund and that it's OK if the memory request fails. After calling the Toolbox routine `NewHandle()` to get the memory, the second `RequestMemory()` sets the flags back to their normal state. If the memory manager wasn't able to allocate 50,000 bytes, the program handles that error itself.

> **Note:** Be sure you send a `gApplication->RequestMemory(FALSE, FALSE)` message after you allocate memory.

```
long GrowMemory(Size bytesNeeded);
```

The `GrowZoneFunc()` function the Memory Manager calls in low memory situations invokes this method to try to reclaim memory. `GrowMemory()` sends a `MemoryShortage()` message to the application.

If there isn't enough memory, it uses the rainy day fund, but only if the number of bytes needed is less than the credit limit or if the loan approved flag is set. The credit limit is set at application initialization. Use the `RequestMemory()` method to set the loan approved flag.

If there still isn't enough memory, and if the can fail flag is set, this method simply returns, and it is up to your application to handle the failed memory allocation. If the can fail flag is

not set, this method sends an `OutOfMemory()` message to the application. Your application should not override this method in most cases.

```
void MemoryShortage(Size bytesNeeded);
```

The `GrowMemory()` method sends a `MemoryShortage()` message to your application to try to free memory. The default `MemoryShortage()` method does nothing, so your application subclass should override this method. Your `MemoryShortage()` method should try to free `bytesNeeded` bytes of memory, and it should disable menu commands that won't work in low memory situations.

> **Note:** Your `MemoryShortage()` method must not allocate any memory.

```
void MemoryReplenished(void);
```

Your application will get this message when the memory situation is no longer critical. The default `MemoryReplenished()` method does nothing, so your application subclass should override this method. Your `MemoryReplenished()` method should enable the commands you disabled in the `MemoryShortage()` method. You might also want to reallocate memory that you released in that method.

```
long OutOfMemory(Size bytesNeeded);
```

A memory request cannot be satisfied, and the can fail flag was not set, so the application won't handle the memory allocation failure. If there is enough memory, this method sends the global error handler (stored in the global variable `gError`) a `CheckOSError(memFullErr)` message and jumps to the beginning of the event loop.

You can override this method if you can free up more memory, or if you want to exit the application gracefully. Normally, you would try to free enough memory in your `MemoryShortage()` method. If there are more drastic ways to release memory, you would implement them here. This method should return 1L if it successfully released memory or zero if it couldn't.

> **Note:** This method should not allocate any memory or call any routines that allocate memory.

Execution methods

The execution methods are invoked while your application is running. Most of these methods handle system-related events. The only execution method your application should override is the `Exit()` method.

```
void Run(void);
```

This method runs your application until the user quits. This method sends `ProcessEvent()` messages to the switchboard (stored in the instance variable `itsSwitchboard`) until the user chooses to quit. This is the method that implements the standard Macintosh event loop.

Before running your program, this method sends a `Preload()` message to your application to open or print documents that the user selected and opened from the Finder.

If you installed an urgent chore, this method sends it a `Perform()` message. Your application should not override this method.

```
void Preload(void);
```

If the user opened or chose to print files from the Finder, this method sends the application an `OpenDocument()` message for each document. If the user chose the **Print** command, this method sends a `DoCommand(cmdPrint)` message to the gopher. After processing all the files, this method sends the application a `StartUpAction()` message.

```
void StartUpAction(short numPreloads);
```

This method gives you an opportunity to perform any startup actions. `NumPreloads` is the number of files that the user selected from the Finder. If `numPreloads` is zero, the default method sends a `DoCommand(cmdNew)` message to the gopher (usually the application). The effect of the default method is to open an untitled document at startup.

```
void Suspend(void);
```

Your application is about to be suspended under MultiFinder. The default method sends a `Suspend()` message to each of the application's directors and sets the global variable `gInBackground` to TRUE.

```
void Resume(void);
```

Your application has come back to the foreground under MultiFinder. The default method sends a `Resume()` message to each of the application's directors and sets the global variable `gInBackground` to FALSE.

```
void SwitchToDA(void);
```

A desk accessory is becoming active. The default method sends the application a `Suspend()` message. Your application should not override or use this method.

```
void SwitchFromDA(void);
```

Your application is becoming active after a desk accessory was active. The default method sends your application a `Resume()` message. Your application should not override or use this method.

```
void Idle(EventRecord *macEvent);
```

This method handles periodic tasks. It also checks to see if a critical memory situation is no longer critical. `Idle()` sends `Dawdle()` messages to the gopher and to each of the gopher's supervisors. It also sends `Perform()` messages to all chores. Your application should not override this method.

```
void Quit(void);
```

The user wants to quit the application. This method sends a `Quit()` message to each of the application's directors. Any of the directors can abort quitting. Your application should not override this method.

> **Note:** The `Quit()` method for directors returns a Boolean value. If it returns FALSE, quitting is aborted.

```
void Exit(void);
```

The application is about to exit. Your main program should send an `Exit()` message to the application before it terminates; the THINK Class Library will not send the message for you.

The `Exit()` method is the last chance you have to clean up after your application. For example, this is a good place to delete any temporary files you've created.

```
void JumpToEventLoop(void);
```

If you send a `JumpToEventLoop()` message to your application, it will start again at the beginning of the event loop. This method is useful for getting out of errors. The `SevereMacError()` method in the CError class uses this method. Your application should not override this method.

> **Note:** You can use the `f_SetJump()` and `f_LongJump()` routines to implement your own error recovery methods.

Document methods

The document methods of an application deal with creating and opening documents. In your application subclass, the only document methods you'll need to override are `CreateDocument()` and `OpenDocument()`. All the other document methods are used internally to implement standard behavior.

```
void CreateDocument(void);
```

This method creates a new document. The default `DoCommand()` method sends a `CreateDocument()` message to the application when the user chooses **New** from the **File** menu. Your application must override this method. Your `CreateDocument()` method needs to create a new document, initialize it, and then send it a `NewFile()` message.

```
void OpenDocument(SFReply *macSFReply);
```

This method opens an existing document. The `macSFReply` record specifies which document to open. The default `DoCommand()` method sends an `OpenDocument()` message to the application when the user chooses **Open...** from the **File** menu.

> **Note:** `DoCommand()` actually sends a `ChooseFile()` message to the application first, and if the reply is good, it sends an `OpenDocument()` mes-

sage. Your `OpenDocument()` method can assume that the `macSFReply` record is valid.

Your application subclass must override this method. Your `OpenDocument()` method needs to create a new document, initialize it, and then send it an `OpenFile(macSFReply)` message.

```
void ChooseFile(SFReply *macSFReply);
```

This method displays a standard get file dialog and places the reply in the `macSFReply` record. This method calls the Toolbox routine `SFPGetFile()` with the file parameters you specified in the `SetUpFileParameters()` method.

You can send the application a `ChooseFile()` message from other methods to get the name and location of a file. Don't forget to check the `macSFReply.good` field to make sure that the information in the record is valid.

Your application subclass should not override this method unless it uses a different technique to open a document.

```
void AddDirector(CDirector *aDirector)
```

This method adds a director (typically a document) to the application's list of directors. The default initialization method for directors sends this message to the application. You should not override or use this method.

```
void RemoveDirector(CDirector *theDirector);
```

This method removes a director (typically a document) to the application's list of directors. The default `Dispose()` method for directors sends this message to the application. You should not override or use this method.

Periodic task methods

```
void AssignIdleChore(CChore *theChore);
```

This method adds `theChore` to the application's list of chores. The application sends a `Perform()` message to each chore at idle time. Your application should not override this method.

> **Note:** If you want to remove the chore from the list later on, you must keep a pointer to the chore object.

```
void CancelIdleChore(CChore *theChore);
```

This method removes `theChore` from the application's list of chores. Your application should not override this method.

```
void AssignUrgentChore(CChore *theChore);
```

This method adds `theChore` to the application's list of urgent chores. The application sends a `Perform()` message to each urgent chore after handling the current event and then removes the chore. Your application should not override this method.

Class resources

`MBAR 1`	The application's menu bar
`STR# 129`	Low memory warning strings.

CApplication summary

```
struct CApplication : CBureaucrat {
    CSwitchboard    *itsSwitchboard;
    CCluster        *itsDirectors;
    CList           *itsIdleChores;
    CCluster        *itsUrgentChores;
    Boolean         urgentsToDo;
    Boolean         running;
    Size            rainyDayFund;
    Handle          rainyDay;
    Boolean         rainyDayUsed;
    short           sfNumTypes;
    SFTypeList      sfFileTypes;
    ProcPtr         sfFileFilter;
    ProcPtr         sfGetDLOGHook;
    short           sfGetDLOGid;
    ProcPtr         sfGetDLOGFilter;
    JumpBuffer      eventLoopJump;
    Boolean         memWarningIssued;
    Size            creditLimit;
    Boolean         loanAPproved;
    Boolean         canFail;

    void      IApplication(short extraMasters, Size aRainyDayFund, Size aCreditLimit);
    void      InitToolbox(void);
    void      InitMemory(short extraMasters, Size aRainyDayFund, Size aCreditLimit);
    void      MakeDesktop(void);
    void      MakeClipboard(void);
    void      MakeDecorator(void);
    void      SetUpFileParameters(void);
    void      SetUpMenus(void);

    void      Notify(CTask *theTask);
    void      DoKeyDown(char theChar, Byte keyCode, EventRecord *macEvent);
    void      DoAutoKey(char theChar, Byte keyCode, EventRecord *macEvent);
    void      DoKeyUp(char theChar, Byte keyCode, EventRecord *macEvent);
    void      DoCommand(long theCommand);
    void      UpdateMenus(void);

    void      RequestMemory(Boolean aLoanApproved, Boolean aCanFail);
    long      GrowMemory(Size bytesNeeded);
    void      MemoryShortage(Size bytesNeeded);
    void      MemoryReplenished(void);
    long      OutOfMemory(Size bytesNeeded);

    void      Run(void);
    void      Preload(void);
```

```
    void        StartUpAction(short numPreloads);
    void        Suspend(void);
    void        Resume(void);
    void        SwitchToDA(void);
    void        SwitchFromDA(void);
    void        Idle(EventRecord *macEvent);
    void        Quit(void);
    void        Exit(void);
    void        JumpToEventLoop(void);

    void        CreateDocument(void);
    void        OpenDocument(SFReply *macSFReply);
    void        ChooseFile(SFReply *macSFReply);
    void        AddDirector(CDirector *aDirector);
    void        RemoveDirector(CDirector *theDirector);

    void        AssignIdleChore(CChore *theChore);
    void        CancelIdleChore(CChore *theChore);
    void        AssignUrgentChore(CChore *theChore);
};
```

CBartender
18

Introduction

The bartender is the object that manages the menu bar, menus, and menu items. The bartender is an object of class CBartender. There is only one object of this class, and it's stored in a global variable.

Heritage

Superclass	CObject
Subclasses	This class has no subclasses. This class should not have any subclasses.

Using CBartender

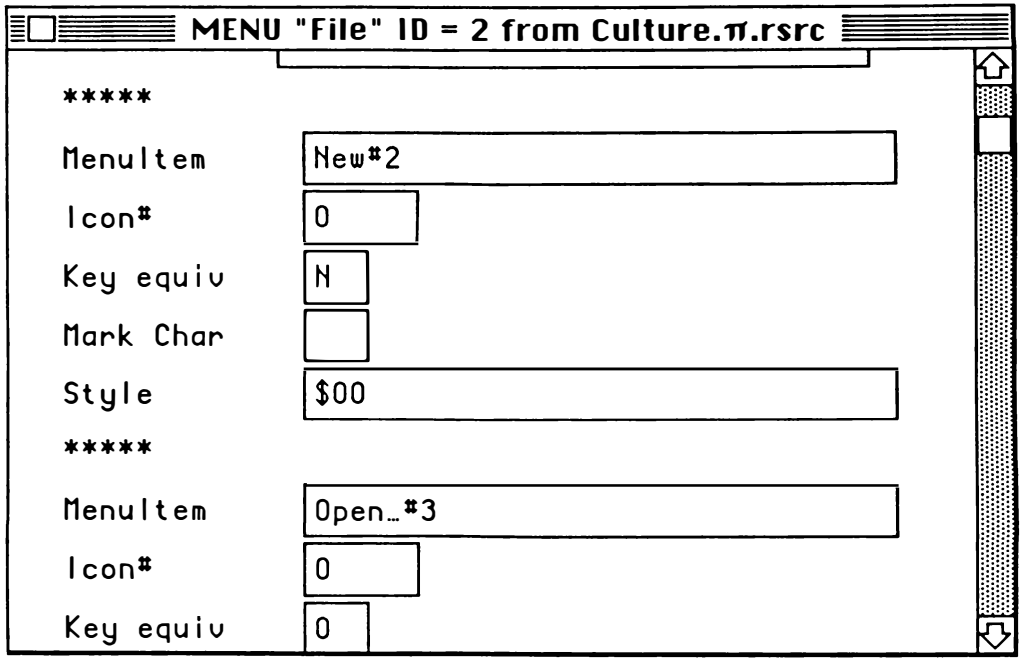
In the THINK Class Library, almost every menu item has a command number associated with it. The bartender maintains a table that maps menu items with command numbers. You should never try to access the bartender's data structures directly. Instead, use the access methods.

You should not need to override the CBartender class. If, for some reason, you do need to override this class, you'll also need to override the `SetUpMenus()` method in your application class to initialize your bartender.

To manipulate the menus or menu items, send messages to the global bartender object stored in the global variable `gBartender`.

Creating standard menus

To associate a command number with a menu item, append the command number to the end of the menu item in your MENU resource. The menu item text and the command number are separated by the character #. This is what part of the **File** menu looks like in ResEdit:



If you don't append a command number to a menu item, the bartender automatically associates the command number `cmdNull` with it.

The bartender builds its tables from the information in the MBAR resource you pass to its initialization method. Your application's MBAR resource should contain the menu IDs of all the menus that will appear in the menu bar.

> **Note:** By default, the application's `SetUpMenus()` method uses the MBAR resource with an ID = 1.

You can use any menu ID for your application's menus. The THINK Class Library reserves the following menu IDs for certain menus:

Menu title	Menu ID	Mnemonic
Apple	1	`MENUapple`
File	2	`MENUfile`
Edit	3	`MENUedit`
Font	10	`MENUfont`
Size	11	`MENUsize`

> **Note:** The `MENU` resources for these menus are in the file `TCL Resources`.

If you want the bartender to return the menu ID and item number of a particular menu item, use the special command -1 in your `MENU` resource. The bartender will return the negative of the menu ID in the high word and the menu item number in the low word. This is the same as menus that you build on the fly, described next.

Resource based menus

Your application may use menus that don't have menu commands associated with each item. The most common example is a **Font** menu. Use the reserved **Font** menu in your `MBAR` resource, and add the resources with the `AddResMenu()` Toolbox routine. Here's how you might write your `SetUpMenus()` method to include a font menu.

```
void CYourApp::SetUpMenus()
{
   MenuHandle     macMenu;

   inherited::SetUpMenus();
   AddResMenu(GetMHandle(MENUfont), 'FONT');
   SetDimOption(MENUfont, dimNONE);
   SetUnchecking(MENUfont, TRUE);
}
```

> **Note:** To learn about `SetDimOption()` and `SetUnchecking()`, see "Dimming and checking menu items" below.

When you build a menu on the fly like this, the bartender does not associate a command number with the menu items. In this case, the bartender's `FindCmdNumber()` returns the negative of the menu ID in the high word and the item ID in the low word.

For example, if you choose the ninth font in the **Font** menu, `FindCmdNumber()` returns -655351 (`0xFFF60009`). Since the number is negative, you know that there is no command number associated with the item. All you have to do is split up the return value into high

word and low word. The high word, `0xFFF6`, is -10, and the low word, `0x0009`, is 9, which means the 9th item of the menu with ID 10.

> **Note:** You can use the macros `HiShort()` and `LoShort()` (defined in `Global.h`) to extract the high word and low word of a long integer.

You can add menu items to existing menus if you like. You might want to add the **Font** menu to a general text-handling menu, or you might want to have a menu with the names of all the documents your application has opened. The important thing to remember is to add all these menu items at the **end** of the existing menu. Otherwise, the bartender will get confused.

Creating hierarchical menus

Most of the time you'll be able to set up your hierarchical menus in your resource file. The key thing to remember is that the item the hierarchical menu is attached to uses the command key equivalent and the item mark differently. The command key value must be 0x1B (control-[), and the value of the item mark is the resource ID of the hierarchical menu.

> **Note:** For a detailed description of hierarchical menus, see *Inside Macintosh V*, Chapter 13, "The Menu Manager."

Sometimes, you can't specify a hierarchical menu in a resource. In these cases, use the `InsertHierMenu()` method. This method adds the menu to a menu in the menu bar and to the bartender's table. Generally, you'll specify hierarchical menus in your resource file. Use this method to insert hierarchical menus from your program.

For hierarchical menus to work correctly, you must set them up either in the resource file or with a call to `InsertHierMenu()`. If you use only the Macintosh Toolbox routines, the bartender's item checking methods will not work.

This example shows how you'd insert a hierarchical **Font** menu as the third item in a **Text** menu from your program.

```
void  CYourApp::SetUpMenus()
{
   MenuHandle      macMenu;
   short           MENUtext, itemNo;

   inherited::SetUpMenus();
   gBartender->InsertHierMenu(MENUfont, cmdNull, MENUtext, 2);
   AddResMenu(GetMHandle(MENUfont), 'FONT');
   SetDimOption(MENUfont, dimNONE);
   SetUnchecking(MENUfont, TRUE);
}
```

Dimming and checking menu items

The bartender includes methods to let you enable and disable and check and uncheck menu items. When you click in the menu bar, the bartender sends an `UpdateMenus()` message to all of the bureaucrats in the Chain of Command. In the general case, all the items in the menu start out dimmed and unchecked. Then each bureaucrat enables the menu items that pertain to it. Once the appropriate items have been enabled and checked, the Toolbox routine `MenuSelect()` displays all the menus.

> **Note:** The dimming, undimming, and unchecking take very little time. You won't notice a delay between the time you click on the menu bar and when the menu is displayed.

Suppose you click in the menu bar of a text processing application. When you click on the menu bar, but before the Toolbox displays the menu, the bartender disables all the menu items. Then, the application enables all the application related menu items: **New**, **Open...**, **Quit**, for example. The document enables all the document related items: **Save**, **Save As...**, **Revert** (if the document's been changed), and so on. A pane might check the current font and size in the **Font** menu. Finally the menu appears on the screen with the correct items checked and enabled.

The `UpdateMenus()` method of your application, document, and pane need to enable each item. To make sure that item enabling happens from the general (application) to the specific (pane), be sure to call `inherited::UpdateMenus()` first in your own `UpdateMenus()` method.

You can use the bureaucrat methods `SetDimOption()` and `SetUnchecking()` in your application `SetUpMenus()` method to modify this behavior. `SetDimOption()` lets you specify whether the bartender should dim all, some, or none of the items when you click on the menu bar. For **Font** menus, for instance, it doesn't make sense to dim all the font names only to re-enable them again.

Variables

The global variable `gBartender` points to the single instance of the bartender. Whenever you want to manipulate menus, you'll send messages to this object. The bartender uses its *instance* variables to maintain the mapping between command numbers and menu items. You should not need to access or alter the instance variables.

Global variable

`CBartender *gBartender`	The global bartender.

Instance variables

`short numMenus;`	The number of menus available.
`MenuEntryH theMenus;`	A table with an entry for each available menu.

Methods

To use these methods to manipulate menus and menu items, send messages to the bartender object stored in the global `gBartender`.

For example, to disable the **Open...** command you would write:

```
gBartender->DisableCmd(cmdOpen);
```

To change the name of the **Copy** command to **Copy Picture**, you might write:

```
gBartender->SetCmdText(cmdCopy, "\pCopy Picture");
```

In the rare case where you need the actual menu handle or other information about the menu, use the `FindMacMenu()` or `FindMenuItem()` methods.

Construction methods

`void IBartender(short MBARid);`

Initialize a Bartender object. The application method `SetUpMenus()` sends this message to initialize the bartender and stores the bartender in the global variable `gBartender`. `MBARid` is the ID of the MBAR resource that the bartender uses to build its menu tables. The default `SetUpMenus()` method uses 1 as its `MBARid`.

Insertion and deletion methods

`void AddMenu(short MENUid, Boolean install, short beforeID);`

This method adds a menu to the bartender's list. `MENUid` is the resource ID of the menu. If `install` is TRUE, install it into the menu bar as well as into the bartender's list. `BeforeID` specifies before which menu this menu is installed. If `beforeID` is 0, the menu is added to the end of the menu bar. A `beforeID` of -1 is a hierarchical or pop-up menu.

`void RemoveMenu(short MENUid);`

Remove the specified menu from the menu bar and from the bartender's list. `MENUid` is the resource ID of the menu.

`void InsertInBar(short MENUid, short beforeID);`

Insert a menu in the menu bar and in the bartender's list. This message is the same as `AddMenu(MENUid, TRUE, beforeID)`.

```
void DeleteFromBar(short MENUid);
```
Remove the specified menu from the menu bar, but leave it in the list.

```
void InsertHierMenu(short hMENUid, long cmdNo, short inMENUid,
short afterItem);
```
Insert a hierarchical menu in another menu. `HMENUid` is the menu ID of the hierarchical menu. If you want to be able to dim a hierarchical menu, provide a command number in `cmdNo`; if you're not going to dim the hierarchical menu, pass in `cmdNull` for `cmdNo`. `InMENUid` is the menu ID of the menu that will contain the hierarchical menu. `AfterItem` is the item number after which the hierarchical menu should be inserted. The title of the menu is used as the item name for the hierarchical menu.

Item manipulation methods

```
void EnableCmd(long cmdNo);
```
Enable the menu item associated with `cmdNo`.

```
void DisableCmd(long cmdNo);
```
Disable the menu item associated with `cmdNo`.

```
void EnableMenu(short MENUid);
```
Enable the menu with resource ID `MENUid`.

```
void DisableMenu(short MENUid);
```
Disable the menu with resource ID `MENUid`.

```
void SetCmdText(long cmdNo, Str255 theText);
```
Change the text of the menu item associated with `cmdNo` to `theText`.

```
void GetCmdText(long cmdNo, Str255 theText);
```
Get the text of the menu item associated with `cmdNo`. If `cmdNo` is not associated with a menu item, the contents of `theText` are unchanged.

```
void CheckMarkCmd(long cmdNo, Boolean checked);
```
Place (or remove) a check mark next to the menu item associated with `cmdNo`.

Item insertion and deletion

```
void InsertMenuCmd(long cmdNo, Str255 theText, short MENUid, short
afterItem);
```
Insert a new item in a menu. `CmdNo` is the command number of the new item. `TheText` is the text of the new item. `MENUid` is the menu ID you're inserting the new item into. `AfterItem` specifies after which item to insert the new item.

```
void RemoveMenuCmd(long cmdNo);
```
Remove the menu item associated with the command cmdNo from its menu.

Look-up methods

These methods let you know which command number is associated with which menu item and which menu contains the menu item associated with a particular command number.

```
long FindCmdNumber(short MENUid, short itemNo);
```
Return the command number associated with the menu `MENUid` and `itemNo`. If the item has no command associated with it, this method returns `-MENUid` in the high word and `itemNo` in the low word.

This is the message that the desktop and switchboard send to the bartender to convert a menu selection or a menu key into a command number. Some menu items, like font menus, don't have commands associated with them. For these menus, `FindCmdNumber()` returns the menu id and item number.

```
void FindMenuItem(long cmdNo, short *MENUid, MenuHandle *macMenu,
short *itemNo);
```
Given a command number, return the menu ID, the handle to the menu, and the item number. If the command number isn't associated with a menu item, all three items are set to zero.

```
short FindItemText(short MENUid, Str255 itemStr);
```
Return the item number in `MENUid` with the specified text. Return 0 if the menu doesn't have an item called `itemStr`. This method is useful for keeping track of items that don't have command numbers associated with them.

```
MenuHandle FindMacMenu(short MENUid);
```
Return a handle to the Macintosh menu with ID `MENUid`. If the menu is not in the bartender's table, return NULL.

```
short FindMenuIndex(short MENUid);
```
Return the index of the menu `MENUid` in the bartender's table. This is an internal method. You should not use this method.

Appearance methods

```
void SetDimOption(short MENUid, DimOption aDimming);
```

Set the dim option for the specified menu. By default, all the items in the menu will be dimmed when you click in the menu bar. The `UpdateMenus()` method of each of your bureaucrats needs to enable the appropriate items.

Dim Option	Meaning
`dimNONE`	Never dim any of the menu items.
`dimSOME`	Dim only the menu items that have command numbers associated with them.
`dimALL`	Dim all of the menu items. Each bureaucrat's `UpdateMenus()` method must enable the items for the commands it handles. This is the default.

```
void SetUnchecking(short MENUid, Boolean anUnchecking);
```

Set the checking option for the specified menu. By default, none of the items are unchecked when you click in the menu bar. If you set this option to TRUE, all the items will be unchecked, and your `UpdateMenus()` methods must check the appropriate menu items.

`TRUE`	Uncheck all the menu items at menu selection. Your `UpdateMenus()` method should check the appropriate items. Set this option for menus like **Font** menus or **Style** menus.
`FALSE`	Don't uncheck any menu items at menu selection. This is the default since most menu items never need to be checked.

```
void UpdateAllMenus(void)
```

The bartender gets this message before menu selection. First it disables and unchecks *all the* items in the menus according to the dimming and unchecking options. Then it sends an `UpdateMenus()` message to the gopher to enable the appropriate menu items. You should not use or override this method.

Command Extraction methods

These three methods are internal methods that the bartender uses to find commands associated with menu items and to build its internal data structures. You should not use these methods.

```
void ExtractCommands(MenuEntryP theEntry);
void ParseItemString(Str255 itemStr, long *cmdNo);
void ExtractHierMenus(MenuHandle macMenu, short index);
```

CBartender Summary

```
struct CBartender : CObject {
    short           numMenus;
    MenuEntryH      theMenus;

    void        IBartender(short MBARid);

    void        AddMenu(short MENUid, Boolean install, short beforeID);
    void        RemoveMenu(short MENUid);
    void        InsertInBar(short MENUid, short beforeID);
    void        DeleteFromBar(short MENUid);
    void        InsertHierMenu(short hMENUid, long cmdNo, short inMENUid, short afterItem);

    void        EnableCmd(long cmdNo);
    void        DisableCmd(long cmdNo);
    void        EnableMenu(short MENUid);
    void        DisableMenu(short MENUid);
    void        SetCmdText(long cmdNo, Str255 theText);
    void        GetCmdText(long cmdNo, Str255 theText);
    void        CheckMarkCmd(long cmdNo, Boolean checked);

    void        InsertMenuCmd(long cmdNo, Str255 theText, short MENUid, short afterItem);
    void        RemoveMenuCmd(long cmdNo);

    short           FindMenuIndex(short MENUid);
    MenuHandle      FindMacMenu(short MENUid);
    long            FindCmdNumber(short MENUid, short itemNo);
    void            FindMenuItem(long cmdNo, short *MENUid, MenuHandle *macMenu,
                                 short *itemNo);
    short           FindItemText(short MENUid, Str255 itemStr);

    void        ExtractCommands(MenuEntryP theEntry);
    void        ParseItemString(Str255 itemStr, long *cmdNo);
    void        ExtractHierMenus(MenuHandle macMenu, short index);

    void        SetDimOption(short MENUid, DimOption aDimming);
    void        SetUnchecking(short MENUid, Boolean anUnchecking);
    void        UpdateAllMenus(void);
};
```

These are the menu IDs that the THINK Class Library reserves. They're defined in `Commands.h`.

```
#define     MENUapple   1     /* Apple menu with DA's, etc. */
#define     MENUfile    2     /* File menu with New, Open, etc. */
#define     MENUedit    3     /* Edit menu with Cut, Copy, etc. */
#define     MENUfont    10    /* Menu of installed fonts */
#define     MENUsize    11    /* Menu of font sizes */
```

These are the commands that the THINK Class Library defines. They're in the file Commands.h. Remember that the THINK Class Library reserves the commands in the range 1 to 1023.

```
#define       cmdNull           0L    /* Command which does nothing      */

#define       cmdOK             100L  /* OK button in dialog box         */
#define       cmdCancel         101L  /* Cancel button in dialog box     */

#define       cmdAbout          256L  /* About Application request       */

    /** Standard File Menu commands **/
#define       cmdQuit           1L
#define       cmdNew            2L
#define       cmdOpen           3L
#define       cmdClose          4L
#define       cmdSave           5L
#define       cmdSaveAs         6L
#define       cmdRevert         7L
#define       cmdPageSetup      8L
#define       cmdPrint          9L

    /** Standard Edit Menu commands **/
#define       cmdUndo           16L
#define       cmdCut            18L
#define       cmdCopy           19L
#define       cmdPaste          20L
#define       cmdClear          21L
#define       cmdToggleClip     22L   /* Show or Hide the Clipboard window */

    /** Text Styles **/
#define       cmdPlain          30L
#define       cmdBold           31L
#define       cmdItalic         32L
#define       cmdUnderline      33L
#define       cmdOutline        34L
#define       cmdShadow         35L
#define       cmdCondense       36L
#define       cmdExtend         37L

    /** Text Alignment **/
#define       cmdAlignRight     40L
#define       cmdAlignLeft      41L
#define       cmdAlignCenter    42L
#define       cmdJustify        43L

    /** Line Spacing **/
#define       cmdSingleSpace    50L
#define       cmd1HalfSpace     51L
#define       cmdDoubleSpace    52L
```

CBorder
19

Introduction

CBorder is a pane with a border.

Heritage

Superclass	CPane
Subclasses	None

Using CBorder

CBorder is a pane with a border. If you want to put a border around a pane, create a border instance and make it your pane's enclosure. The thickness of a border is the width of the line that frames the border. The default thickness is 1. A border can also have a drop shadow. The default size for a drop shadow is 0 (no drop shadow).

Variables

`short thickness;`	Thickness of the border.
`short dropShadow;`	Size of the drop shadow.

Methods

Creation and destruction

```
void IBorder(CView *anEnclosure, CBureaucrat *aSupervisor, short
aWidth, short aHeight, short aHEncl, short aVEncl, SizingOption
aHSizing, SizingOption aVSizing);
```

Initialize a border.

> **Note:** The descriptions of the other arguments are in CPane.

```
void IViewRes(ResType rType, short resID, CView *anEnclosure,
CBureaucrat *aSupervisor);
```

Initialize a border from a resource template. `RTtype` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the same as for `IBorder()`. This method is inherited from CView.

To initialize a border from a resource file, use a `'Bord'` resource.

```
void IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor, Ptr
viewData);
```

This method is used internally for initializing from a resource template. Each subclass of CView overrides this method to use its own resource template.

Accessing methods

```
void SetThickness(short aThickness);
```

Specify the thickness of the border.

```
void SetDropShadow(short aDropShadow);
```

Specify the size of the drop shadow of the border.

```
void SetInteriorSize(short wInterior, short hInterior);
```

Specify the size of the interior of a border. The frame of the border is adjusted so the interior matches the specification. The border is not redrawn.

```
void GetInterior(Rect *theInterior);
```

Get the interior of the border. The interior does not include the thickness of the border or the drop shadow.

Drawing methods

```
void Draw(Rect *area);
```

Draw the border.

CBorder summary

```
struct CBorder : CPane {
    short     thickness;
    short     dropShadow;

    void      IBorder(CView *anEnclosure, CBureaucrat *aSupervisor,
                  short aWidth, short aHeight,
                  short aHEncl, short aVEncl,
                  SizingOption aHSizing, SizingOption aVSizing);
    void      IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor, Ptr viewData);

    void      SetThickness(short aThickness);
    void      SetDropShadow(short aDropShadow);
    void      SetInteriorSize(short wInterior, short hInterior);
    void      GetInterior(Rect *theInterior);

    void      Draw(Rect *area);
};
```

CBureaucrat 20

Introduction

CBureaucrat is an abstract class that implements a link in the chain of command. Any object in the THINK Class Library that responds to a menu selection or a mouse click is a bureaucrat.

A bureaucrat can either respond to a command, or it can pass the command to its supervisor. All of the default methods for bureaucrats pass the message on to the supervisor, so if a particular subclass cannot handle the message, calling the inherited method will cause the message to be passed on up the chain of command.

Heritage

Superclass	CObject
Subclasses	CApplication
	CView
	CDirector
	CRadioGroup

Using CBureaucrat

You generally won't need to create a subclass of CBureaucrat because most of the objects you'll use are already descendants of this class. However, you might find that you need to create a class that's a link in the chain of command that's not one of the common objects. An example of this kind of class is CRadioGroup which acts as the supervisor for a collection of radio buttons.

Variables

Global variable

The global variable `gGopher` points to the current bureaucrat which is the first object in the chain of command. Whenever the switchboard sends a `DoCommand()` message, it sends it to the gopher. If the object `gGopher` points to cannot handle the command, it passes the `DoCommand()` message up the chain of command until the message reaches an object that handles the command or until it reaches the end of the chain of command— the application.

When there are no documents open, gGopher points to the application. When you open a document, gGopher points to the document. Your document subclass may change gGopher to point to the main pane.

```
CBureaucrat *gGopher;
```
The current bureaucrat.

Instance variable

A bureaucrat has only one instance variable: its supervisor.

```
CBureaucrat *itsSupervisor;
```
The bureaucrat's supervisor.

Methods

Construction and destruction methods

```
void IBureaucrat(CBureaucrat *aSupervisor);
```
Initialize the bureaucrat. This method sets aSupervisor to be itsSupervisor

```
void Dispose(void);
```
If the bureaucrat is the gopher, set the global variable gGopher to the bureaucrat's supervisor before disposing of the object.

Accessing method

```
CBureaucrat *GetSupervisor(void);
```
Return the bureaucrat's supervisor

Command methods

These are the messages that every bureaucrat accepts. Unless a bureaucrat (or one of its ancestors) implements one of these methods, it will just pass the message on to its supervisor. If you follow the chain of supervisors— the chain of command— all the way to the end, you'll end up at your application class.

```
void Notify(CTask *theTask);
```
A task has been completed. This method passes the Notify() message to the bureaucrat's supervisor.

```
void DoKeyDown(char theChar, Byte keyCode, EventRecord *macEvent);
```
Handle a key-down event. This method passes the DoKeyDown() message to the bureaucrat's supervisor.

```
void DoAutoKey(char theChar, Byte keyCode, EventRecord *macEvent);
```
Handle an auto-key event. This method passes the DoAutoKey() message to the bureaucrat's supervisor.

```
void DoKeyUp(char theChar, Byte keyCode, EventRecord *macEvent);
```

Handle a key-up event. This method passes the `DoKeyUp()` message to the bureaucrat's supervisor.

Note: By default, the Toolbox Event Manager masks out key up events.

```
void DoCommand(long theCommand);
```

Handle a command. This method passes the `DoCommand()` message to the bureaucrat's supervisor.

```
void Dawdle(long *maxSleep);
```

Perform periodic actions at idle time. The default method does nothing. CApplication's `Idle()` method sends a `Dawdle()` message to all of the bureaucrats in the chain of command.

If your bureaucrat requires periodic actions, perform them in the `Dawdle()` method. Set the value of `maxSleep` to the largest number of ticks that your application can tolerate between `Dawdle()` messages. If your application's `Dawdle()` message doesn't have any time constraints, you can ignore `maxSleep`.

The application class uses the value of `maxSleep` to let `WaitNextEvent()` know that it should yield an event after at most `maxSleep` ticks. For instance, the `Dawdle()` method for CEditText calls `TEIdle()` to blink the insertion point and sets `maxSleep` to `GetCaretTime()` which is the rate at which the insertion point blinks.

A blinking insertion point is a good example. A clock display is another good example. You would update the display from your document's `Dawdle()` method. If your clock displays seconds, set `maxSleep` to 60 since you need to update the display at least every 60 ticks.

On a null event, CApplication's `Idle()` method sends a `Dawdle()` message to every bureaucrat in the chain of command. `Idle()` sets the global variable `gSleepTime` to the smallest sleep time that all bureaucrats requested. The switchboard uses this global variable in its call to `WaitNextEvent()`.

```
void UpdateMenus(void);
```

Update the menus that have to do with this bureaucrat. Your bureaucrat (usually a document or a pane) should override this method to enable the menu items that pertain to it. The important thing to remember when you write an `UpdateMenus()` method is to call `inherited::UpdateMenus()` before you enable your own menus. In this way, your supervisor's menus will be updated first. See CBartender for a discussion of menu updating.

CBureaucrat Summary

```
struct CBureaucrat : CObject {

    CBureaucrat     *itsSupervisor;

    void        IBureaucrat(CBureaucrat *aSupervisor);
    void        Dispose(void);

    CBureaucrat     *GetSupervisor(void);

    void        Notify(CTask *theTask);
    void        DoKeyDown(char theChar, Byte keyCode, EventRecord *macEvent);
    void        DoAutoKey(char theChar, Byte keyCode, EventRecord *macEvent);
    void        DoKeyUp(char theChar, Byte keyCode, EventRecord *macEvent);
    void        DoCommand(long theCommand);
    void        Dawdle(long *maxSleep);
    void        UpdateMenus(void);
};
```

CButton
21

Introduction

CButton implements a standard Macintosh push button. You can use a button in any pane or window.

Heritage

Superclass	CControl
Subclasses	CCheckBox
	CRadioButton

Using CButton

CButton implements a standard Macintosh push button. You should not override this class unless you want to implement a push button that works differently from the default button. (The other kinds of Macintosh buttons, check boxes and radio buttons, are subclasses of this class.)

The CButton class handles all the mouse tracking for you. When you click and release the mouse button within a button, the button sends a `DoCommand()` message to its supervisor. To specify which command number the button will send in the `DoCommand()` message, send it a `SetClickCmd()` message.

This is what happens when you click in a button. The `DoClick()` method that CButton inherits from CControl calls the Macintosh Toolbox routine `TrackControl()` to highlight the button and track mouse movement. When you release the mouse within a button, the `DoClick()` method sends the control a `DoGoodClick()` message. The `DoGoodClick()` method sends the `DoCommand()` message to the button's supervisor.

Variables

`long clickCmd;`	Command number to send after a click. The default is `cmdNull`.

Methods

Although CButton implements only these methods you can use the methods it inherits from CControl and CView to manipulate a button's title and location.

```
void IButton(short CNTLid, CView *anEnclosure, CBureaucrat
*aSupervisor);
```

Initialize a button. `CNTLid` is the resource ID of a `CNTL` resource (a control template). `AnEnclosure` is the pane or window that the control belongs to. `ASupervisor` is the control's supervisor, typically a document.

`IButton()` places the control at the location specified in the `CNTL` resource. If you want to position the button from your program, send it a `Place()` message. (Controls inherit the `Place()` method from CPane.)

```
void SetClickCmd(long aClickCmd);
```

Set the command number to be sent when the user clicks in a button.

```
void DoGoodClick(short whichPart);
```

When you press and release the mouse within the button, this method sends a `DoCommand()` message to its supervisor. This is an internal method. You should not use or override this method.

CButton summary

```
struct CButton : CControl {
    /** Instance Variables **/
    long        clickCmd;

    void        IButton(short CNTLid, CView *anEnclosure, CBureaucrat *aSupervisor);

    void        DoGoodClick(short whichPart);
    void        SetClickCmd(long aClickCmd);
};
```

The CButton class uses these constants when it returns the value of a button. For push buttons, they don't really make sense since push buttons aren't normally supposed to retain a value. They're used by the descendants of CButton: CCheckBox and CRadioButton.

```
#define     BUTTON_OFF      0
#define     BUTTON_ON       1
```

CCheckBox 22

Introduction

CCheckBox implements a standard Macintosh check box. You can use a check box in any pane or window.

Heritage

Superclass	CButton
Subclasses	None

Using CCheckBox

CCheckBox implements a standard Macintosh check box. You should not override this class unless you want to implement a check box that works differently from the default check box.

The CCheckBox class handles all the mouse tracking for you. When you click and release the mouse button within a check box, it toggles the value of the check box and sends a `DoCommand()` message to its supervisor. To specify which command number the check box will send in the `DoCommand()` message, send it a `SetClickCmd()` message.

> **Note:** CCheckBox inherits the `SetClickCmd()` method from CButton.

This is what happens when you click in a check box. The `DoClick()` method that CCheckBox inherits from CControl calls the Macintosh Toolbox routine `TrackControl()` to highlight the check box and track mouse movement. When you release the mouse within a check box, the `DoClick()` method sends the control a `DoGoodClick()` message. The `DoGoodClick()` method toggles the value of the control and sends the `DoCommand()` message to the check box's supervisor.

To find out whether the check box is checked, send it an `IsChecked()` message. You can set the value of the check box from your program by sending it a `SetValue()` message. Use the values `BUTTON_ON` and `BUTTON_OFF` to specify whether to check or uncheck the check box.

> **Note:** Setting the value with `SetValue()` will not send a `DoCommand()` message to the check box's supervisor.

Variables

This class has no instance variables.

Methods

Although CCheckBox implements only these methods you can use the methods it inherits from CControl and CView to manipulate a check box's title and location.

You should use the `SetClickCmd()` method CCheckBox inherits from CButton to specify a command the check box sends to its supervisor when you click in it.

```
void ICheckBox(short CNTLid, CView *anEnclosure, CBureaucrat
*aSupervisor);
```

Initialize a check box. `CNTLid` is the resource ID of a `CNTL` resource (a control template). `AnEnclosure` is the pane or window that the control belongs to. `ASupervisor` is the control's supervisor, typically a document.

`ICheckBox()` places the control at the location specified in the `CNTL` resource. If you want to position the check box from your program, send it a `Place()` message. (Controls inherit the `Place()` method from CPane.)

```
void DoGoodClick(short whichPart);
```

When you press and release the mouse within the check box, this method toggles the state of the check box and sends a `DoCommand()` message to its supervisor.

This is an internal method. You should not use or override this method.

```
Boolean IsChecked(void);
```

Returns TRUE if the check box is checked.

CCheckBox summary

```
struct CCheckBox : CButton {
    /* None instance variables*/

    void        ICheckBox(short CNTLid, CView *anEnclosure, CBureaucrat *aSupervisor);
    void        DoGoodClick(short whichPart);
    Boolean     IsChecked(void);
};
```

You can use these constants as arguments to the `SetValue()` method to set the state of the check box.

```
#define       BUTTON_OFF        0
#define       BUTTON_ON         1
```

CChore
23

Introduction

CChore is an abstract class for implementing periodic or urgent actions. The THINK Class Library executes periodic chores at idle time and urgent chores after processing the current event.

Heritage

Superclass	CObject
Subclasses	You must define a subclass of this class.

Using CChore

You can use this class to create chores that run at idle time or that run at the next possible opportunity. Your application maintains a list of chores and implements the methods to install and remove them.

To implement a chore, you'll first need to define a subclass of this class. In your class, you'll override the `Perform()` method. This method implements the action you want it to take.

Idle chores

To install an idle chore, send it to the application in an `AssignIdleChore()` message. Your application will send a `Perform()` message to all the idle chores at idle time. To remove an idle chore, send the chore you want to remove to the application in a `RemoveIdleChore()` message. For example:

```
gApplication->AssignIdleChore(theChore);
```

> **Note:** To remove an idle chore, you'll have to keep it in a variable since `RemoveIdleChore()` requires a reference to a CChore object.

Urgent chores

An urgent chore is one that gets executed immediately after the application processes the current event. Typically, it's the same event that caused the urgent chore to be installed. Urgent chores are executed only once. They're removed from the urgent chore list immediately after they're executed.

To install an urgent chore, send it to the application in an `AssignUrgentChore()` message.

Using chores

Note that chores are attached to the application and not to any particular document. The things that a chore does, then, should pertain to the entire application. For example, you could use a chore to communicate between loosely related objects.

If you need your chore to run at a specific interval, you can use the Macintosh Toolbox routine `Ticks()` to store the last time the chore was performed.

All bureaucrats, which includes documents and panes, inherit a `Dawdle()` method. The application sends a `Dawdle()` message to each bureaucrat in the chain of command. For instance, you would use a `Dawdle()` method to blink an insertion point in a pane.

Variables

This class has no instance variables. Your CChore subclass may need to define its own instance variables. For example, if you want your idle chore to run at specified intervals, you could use an instance variable to store the period.

Methods

CChore *implements* only one method which you must override. Your CChore subclass may implement additional methods.

`void Perform(long *maxSleep)`

This is the method that executes your chore. You must override this method in your CChore subclass. The default method does nothing.

The `maxSleep` parameter specifies how many ticks your chore can tolerate before being sent a `Perform()` message. If you need to do a chore at least once every tenth of a second, you would set `maxSleep` to 6. (Each tick is one-sixtieth of a second.)

For a more detailed discussion of `maxSleep`, see the description of the `Dawdle()` method in CBureaucrat.

CChore summary

```
struct CChore : CObject {
/** No instance Variables **/

   void         Perform(long *maxSleep);
};
```

CClipboard 24

Introduction

CClipboard implements a standard Macintosh clipboard, or scrap. The default clipboard handles `TEXT` and `PICT` scraps. To deal with other kinds of scraps or to implement a private scrap, you'll need to create a subclass of this class.

Heritage

Superclass	CDirector
Subclasses	To implement a private scrap, you should define a subclass.

Using CClipboard

This class implements a standard Macintosh clipboard. CClipboard uses the desk scrap to store its data and supports a window to display the contents of the scrap. This implementation supports only `TEXT` and `PICT` data.

When you initialize your application, the default initialization method, `IApplication()` sends a `MakeClipboard()` message. This method creates an instance of CClipboard and stores it in the global variable `gClipboard`.

The default application `DoCommand()` method handles the `cmdToggleClip` command which shows and hides the Clipboard window. When you're running under MultiFinder, the Clipboard window is hidden when your application is suspended. When the application resumes, the Clipboard window becomes visible again.

> **Note:** In the THINK Class Library, desk accessories behave as if they were in their own layer even if your application isn't running under MultiFinder. When a desk accessory window is frontmost, the Clipboard window will be hidden. When one of your application windows is frontmost, the Clipboard window will be visible again.

Implementing a CClipboard subclass

If you want your application to support a private scrap, or if you want to display other types of data, you'll need to implement a subclass of CClipboard. You'll also need to override the `MakeClipboard()` method in your application class.

> **Note:** If you create a subclass of this class, be sure you understand how the Macintosh scrap mechanism works. See *Inside Macintosh I*, Chapter 15, "The Scrap Manager."

In your CClipboard subclass you'll need to override these methods:

Method	Action
`PutData()`	Place data in the private scrap.
`GetData()`	Retrieve data from the private scrap.
`ConvertGlobal()`	Place the contents of the desk scrap into the private scrap.
`ConvertPrivate()`	Place the contents of the private scrap into the desk scrap.
`UpdateDisplay()`	Display the contents of the private scrap.

Variables

The global variable `gClipboard` contains a pointer to the single instance of the clipboard object. The value of this variable is set in the application method `MakeClipboard()`.

Global variable

`CClipboard *gClipboard;`	The global clipboard.

Instance variables

`CPanorama *itsContents;`	Pane for displaying contents
`CScrollPane *itsScrollPane;`	Contents can be scrolled
`long theLength;`	Length from last Get operation
`long theOffset;`	Offset from last Get operation
`short lastScrapCount;`	Count at the last conversion between global and private scraps.
`Boolean privateNewer;`	TRUE if the private scrap has changed since the last conversion to the desk scrap.
`Boolean windowVisible;`	TRUE if the Clipboard window is visible.

Methods

Construction and destruction methods

```
void IClipboard(CBureaucrat *aSupervisor, Boolean hasWindow);
```

Initialize the clipboard. The application's `MakeClipboard()` method sends this message and stores the clipboard object in the global variable `gClipboard`.

> **Note:** If you override this class, you might want to name the initialization method `IClipboard()` as well. Otherwise you'll have to override the `MakeClipboard()` method in the application class as well.

Suspend and resume methods

These methods handle scrap conversion when your application moves from foreground to background under MultiFinder. In most cases, you won't need to use or override these methods.

```
void Suspend(void);
```

The application is about to be switched into the background under MultiFinder. If the application's private scrap is newer than the desk scrap, this method sends `ConvertPrivate()` and `ScrapConverted()` messages to the clipboard. This method then sends the clipboard window a `HideSuspend()` message.

```
void Resume(void);
```

The application is about to be brought to the foreground under MultiFinder. If the desk scrap is newer than the private scrap, this method sends the clipboard `ConvertGlobal()`, `ScrapConverted()`, and `UpdateDisplay()` messages to update the contents of the clipboard. This method then sends the window a `ShowResume()` message to make it visible.

Appearance methods

These methods handle the appearance of the Clipboard window and how the contents of the clipboard appear in the window. If you implement your own clipboard class, you'll need to override the `UpdateDisplay()` method.

```
void Close(Boolean qutting);
```

The user chose **Close** from the **File** menu. This method sends a `CloseWind()` message.

```
void CloseWind(CWindow *theWindow);
```

The user clicked in the window's close box. This method hides the Clipboard window and changes the text in the menu item.

```
void Toggle(void);
```
This method opens the Clipboard window if it's closed or closes it if it's open. The application's default `DoCommand()` method sends this message.

```
void UpdateDisplay(void);
```
Display the contents of the scrap in the Clipboard window. This method supports only `TEXT` and `PICT` type of data. This method will work correctly with a private scrap, but if you want to display other kinds of data, you'll need to override this method.

Accessing methods

Use these methods to place data in and retrieve data from the desk scrap. If your application implements a private scrap, use the `PutData()` and `GetData()` methods instead.

```
void PutGlobalScrap(ResType theType, Handle theData);
```
Put `theData` of type `theType` into the desk scrap.

```
Boolean GetGlobalScrap(ResType theType, Handle theData);
```
Get the data of type `theType` from the desk scrap and put its data in the block `theData` is a handle to. Return TRUE if the method was able to fulfill the request, FALSE otherwise.

`theData` must be an allocated handle. The size of the allocated memory will grow as needed to fit the data. This is how you would get the `TEXT` data from the desk scrap:

```
Handle myData;

myData = NewHandle(0); /* Create a zero-sized handle */
gClipboard->GetGlobalData('TEXT', myData);
```

```
long DataSize(ResType theType);
```
Return the number of bytes of data in the clipboard of type `theType`. If the clipboard doesn't contain any data of the specified type, this method returns zero.

```
ScrapStatus Status(void);
```
Return the status of the scrap. This method returns a value that describes the relationship between the private scrap and the desk scrap.

Value	Meaning
`PRIVATE_SCRAP_NEWER`	The information in the private scrap is newer than the information in the desk scrap.
`GLOBAL_SCRAP_NEWER`	The information in the desk scrap is newer than the information in the private scrap.
`SCRAPS_THE_SAME`	The information in both scraps is the same.

```
void ScrapConverted(void);
```
Set the internal flags after the scrap has been converted.

Scrap conversion methods

Use the `PutData()` and `GetData()` methods to implement Cut, Copy, and Paste commands for your application. If your application uses only the desk scrap, you can use the `PutGlobalScrap()` and `GetGlobalScrap()` methods.

```
void PutData(ResType theType, Handle theData);
```
Put `theData` of type `theType` into the scrap. The default method puts the data in the desk scrap. If your subclass supports a private scrap, you must override this method. Be sure to send a `PrivateChanged()` message after storing your data in the private scrap.

```
Boolean GetData(ResType theType, Handle *theData);
```
Get the data of type `theType` and put it into a newly created handle. This method returns TRUE if the request was successful, FALSE otherwise. If your application supports a private scrap, you must override this method.

Unlike `GetGlobalScrap()` you should not allocate the handle to `theData`. This method will allocate the memory. This is how you'd get data of type `TEXT`:

```
Handle myData;

gClipboard->GetData('TEXT', &myData);
```

```
void ConvertGlobal(void);
```
Convert data in the desk scrap and put it in the private scrap. The default method does nothing. If your application supports a private scrap, you must override this method.

```
void ConvertPrivate(void);
```
Convert data in the private scrap and put it into the desk scrap. The default method does nothing. If your application supports a private scrap, you must override this method.

```
void PrivateChanged(void);
```
The data in the private scrap has changed. If you implement a private scrap, be sure to send this message to the clipboard after you put new data in your private scrap. This method sets the internal flags and sends an `UpdateDisplay()` message to the clipboard. Your subclass should not override this method.

Class resources

The WIND resource for the Clipboard window is in the file TCL Resources which contains all of the resources the THINK Class Library requires.

WIND 200	Window template for the Clipboard window.

CClipboard summary

```
typedef enum {
    GLOBAL_SCRAP_NEWER,
    PRIVATE_SCRAP_NEWER,
    SCRAPS_THE_SAME
} ScrapStatus;

struct CClipboard : CDirector {
    CPanorama       *itsContents;
    CScrollPane     *itsScrollPane;
    long        theLength;
    long        theOffset;
    short       lastScrapCount;
    Boolean     privateNewer;
    Boolean     windowVisible;

    void        IClipboard(CBureaucrat *aSupervisor, Boolean hasWindow);

    void            Suspend(void);
    void            Resume(void);
    Boolean         Close(Boolean quitting);
    void            CloseWind(CWindow *theWindow);

    void            Toggle(void);
    void            PutGlobalScrap(ResType theType, Handle theData);
    Boolean         GetGlobalScrap(ResType theType, Handle theData);
    ScrapStatus     Status(void);
    void            ScrapConverted(void);
    void            ConvertGlobal(void);
    void            ConvertPrivate(void);
    void            PutData(ResType theType, Handle theData);
    Boolean         GetData(ResType theType, Handle *theData);
    void            PrivateChanged(void);
    void            UpdateDisplay(void);
    long            DataSize(ResType theType);
};
```

CCluster
25

Introduction

CCluster implements an unordered list of objects.

Heritage

Superclass	CCollection
Subclasses	CList

Using CCluster

Use a CCluster object whenever you need to maintain an unordered list of objects. Typically, the elements of the list are object references, though you can store anything you want in a cluster. The default size for an item in a cluster is four bytes, which is large enough to hold a pointer or a handle. Several objects in the THINK Class Library use CCluster objects or descendants of CCluster to maintain lists.

> **Note:** If you need an ordered list of objects, use the CList class instead.

CCluster allocates space for each object reference in **blocks**. By default, a block holds three **slots**. Each slot holds one object reference. CCluster takes care of allocating blocks automatically, though you can change the number of slots per block if you like.

Variables

`short blockSize;`	Number of slots to allocate when more space is needed
`short slots;`	Number of slots allocated for storing objects
`LongHandle items;`	Handle to the items (objects) in the cluster

Methods

In addition to the insertion, deletion, and searching methods, CCluster implements iteration methods that let you apply a function to all the objects in a cluster.

Construction and destruction

```
void ICluster(void);
```
Initialize the cluster.

```
void Dispose(void);
```
Dispose of a cluster, but not the items in it.

```
void DisposeAll(void);
```
Dispose of a cluster and all the items in it. This method sends a `Dispose()` message to every item in the cluster.

```
void DisposeItems(void);
```
Dispose of the items in a cluster, but not the cluster.

```
void SetBlockSize(short aBlockSize);
```
Set the block size to `aBlockSize`.

```
void MoreSlots(void);
```
Allocate `blockSize` more slots in the cluster. The `Add()` method sends this message for you, but you can use it yourself if you need to.

Insertion and deletion

These methods add objects to and remove objects from the cluster. Remember that the items in the cluster are not in any particular order.

```
void Add(CObject *theObject);
```
Add `theObject` to the cluster.

```
void Remove(CObject *theObject);
```
Remove `theObject` from the collection if it is already in the cluster.

Membership

These methods let you determine whether a particular item is in the cluster.

```
Boolean Includes(CObject *theObject);
```
Return TRUE if `theObject` is in the cluster.

```
CObject* FindItem(BooleanFunc);
```
Look for an item that satisfies a function. `BooleanFunc` is a pointer to a function declared like this:

```
Boolean MyBoolFunc(CObject *obj);
```

`FindItem()` applies the function to each item in the list and returns the first one that causes the function to return TRUE.

```
CObject* FindItem1(BooleanFunc, long param);
```

Look for an item that satisfies a function. `BooleanFunc` is a pointer to a function declared like this:

```
Boolean MyBoolFunc(CObject *obj, long param);
```

`FindItem1()` applies the function to each item in the list and returns the first one that causes the function to return TRUE.

```
long Offset(CObject *theObject);
```

Returns the index of `theObject` in the collection. If `theObject` is not in the the collection, `Offset()` returns `BAD_INDEX`. This is an internal method. In most cases you won't need to use this method.

Iteration

These methods let you apply a function to all of the objects in a cluster. The function you apply to the objects in the cluster should not affect the cluster itself.

To apply a message to all the items in a cluster, you need to write a message-sending function. It's up to you to make sure that all the items in the cluster can respond to the message. For example, this is how you send a `Draw()` message to all the items in a cluster:

```
void Perform_Draw(CPane *thePane, Rect *area)
{
    thePane->Draw(area);
}

myCluster->DoForEach1(Perform_Draw, (long)(&myRect));
```

```
void DoForEach(VoidFunc);
```

Apply a function to each object in the collection. `VoidFunc` is a pointer to a function that takes only one argument. The function should be declared like this:

```
void MyFunc(CObject *theObject);
```

```
void DoForEach1(VoidFunc, long);
```

Apply a function to each object in the collection. `VoidFunc` is a pointer to a function that takes two arguments. The function should be declared like this:

```
void MyFunc1(CObject *theObject, long param);
```

CCluster Summary

```
struct CCluster : CCollection {
    short       blockSize;
    short       slots;
    LongHandle      items;

    void        ICluster(void);
    void        Dispose(void);
    void        DisposeAll(void);
    void        DisposeItems(void);


    void        SetBlockSize(short aBlockSize);
    void        Add(CObject *theObject);
    void        Remove(CObject *theObject);
    Boolean     Includes(CObject *theObject);

    void        DoForEach(VoidFunc);
    void        DoForEach1(VoidFunc, long);
    CObject*    FindItem(BooleanFunc);
    CObject*    FindItem1(BooleanFunc, long param);

    long        Offset(CObject *theObject);
    void        MoreSlots(void);
};

#define        SLOT_SIZE   4      /* Size of an item in a collection */
```

This is the value that the `Offset()` method returns if it can't find the specified object.

```
#define        BAD_INDEX   -1L   /* Flag indicating a failed search      */
```

CCollection
26

Introduction

CCollection is an abstract class for implementing collections of things.

Heritage

Superclass	CObject
Subclasses	CCluster

Using CCollection

CCollection is an abstract class that helps you implement collections of things. The class doesn't provide the implementation of the collection. In your subclass, you specify the data strutures that implement the list.

The CCluster class implements an unordered collection of objects. You can use it as a model for implementing your kinds of collections.

Variables

The only instance variable in this abstract class is the number of items in the collection. Your subclass will need to add at least the instance variable that contains or points to the memory that contains the items in your collection.

`long numItems`	The number of items in the collection.

Methods

`void ICollection(void);`
Initialize the collection. The default method sets the initial number of items to 0.

`long GetNumItems(void);`
Return the number of items in a collection

`Boolean IsEmpty(void);`
Return TRUE if the collection has no items.

CCollection summary

```
struct CCollection : CObject {
    long            numItems;

    void            ICollection(void);
    long            GetNumItems(void);
    Boolean         IsEmpty(void);
};
```

CControl
27

Introduction

CControl is an abstract class for implementing Macintosh controls. The two built-in descendant classes, CButton and CScrollBar, implement the standard Macintosh controls.

Heritage

Superclass	CPane
Subclasses	CButton
	CScrollBar

Using CControl

This class describes all the common methods to work with controls. Scroll bars, buttons, check boxes, and radio buttons are descendants of CControl. Although you won't be creating subclasses of CControl (unless you define your own kind of control) you should become familiar with the methods.

Variables

The only instance variable for this class is a handle to the actual Macintosh control.

`ControlHandle macControl;`	Handle to the Macintosh control.

Methods

You can use these methods with any kind of control, but some methods don't really apply to all controls. For example, it's possible to set the value of a push button or to give a title to a scroll bar, but in most cases these actions don't make sense.

Construction and destruction methods

The CControl class does not define an intialization method. Each descendant of CControl defines an initialization method that uses CNTL resources to specify the type of control.

```
void Dispose(void);
```

Dispose of a control.

Accessing methods

```
void SetValue(short aValue);
```
Set the value of a control.

```
short GetValue(void);
```
Get the value of a control

```
void SetMaxValue(short aMaxValue);
```
Set the maximum value a control can have. For push buttons, check boxes, and radio buttons this value should be 1.

```
short GetMaxValue(void);
```
Get the maximum value a control can take on.

```
void SetMinValue(short aMinValue);
```
Set the minimum value a control can have. Note that for push buttons, check boxes, and radio buttons this value should be 0.

```
short GetMinValue(void);
```
Get the minimum value a control can take on.

```
void SetTitle(Str255 aTitle);
```
Set the title of a control. Scroll bars can have titles, but the title is not displayed.

```
void GetTitle(Str255 aTitle);
```
Get the title of a control

```
void SetActionProc(VoidFunc anActionProc);
```
Set the action proc of a control. The action proc is called repeatedly while the mouse is held down in a control.

The control manager distinguishes between an action proc called when the mouse goes down in a moving indicator (like a scroll box) and one called when the mouse goes down in a stationary part of a control. The default `DoClick()` method sends the action proc to the control manager only for mouse hits that are not in a moving indicator.

You must declare the action proc like this:

```
pascal void MyAction(ControlHandle macControl, short whichPart);
```

If you want an action proc called when the mouse is in a moving indicator, you'll need to override the `DoClick()` method.

> **Note:** Be sure you understand action procs and the control manager before you use this method.

Appearance methods

These methods control the appearance of controls on the screen. They hide and show the control, draw it, move it, and change its size.

```
void Hide(void);
```
Hide the control.

```
void Activate(void);
```
Make the control active. The control gets an `Activate()` message automatically when its enclosure gets an `Activate()` message.

```
void Deactivate(void);
```
Make the control inactive. The control gets a `Deactivate()` message automatically when its enclosure gets a `Deactivate()` message.

```
void Offset(short hOffset, short vOffset, Boolean redraw);
```
Move the control by `hOffset`, `vOffset` pixels. If `redraw` is TRUE, redraw the control after the move.

```
void ChangeSize(Rect *delta, Boolean redraw);
```
Change the size of a control. Each component of the rectangle specifies by how many pixels to offset each point. Positive numbers mean down and to the right. Negative numbers mean up and to the left. If `redraw` is TRUE, redraw the control after the change.

```
void Draw(Rect *area);
```
Draw the control. The `area` parameter is ignored.

```
void DrawAll(Rect *area);
```
Draw the control and all its subviews.

```
void Prepare(void);
```
Prepare the port and the coordinate system before drawing. Generally, you don't need to use or override this method.

Click response methods

The CControl class takes care of all the mouse tracking within a control.

```
void DoClick(Point hitPt, short modifierKeys, long when);
```

Handle a click in a control. `HitPt` is the point in frame coordinates. `ModifierKeys` is the same as the modifier field of an event record. `When` is the time that the mouse went down in ticks.

The default method handles both simple controls and controls with moving indicators and behaves a little differently in each case.

If the mouse goes down in a part other than a moving indicator:

- The default method calls the Toolbox routine `TrackControl()` with the action proc you specified with the `SetActionProc()` method.
- If `TrackControl()` returns TRUE (the user clicks and releases the mouse in the same part of the control), this method sends a `DoGoodClick()` message to the control.

If the mouse goes down in a moving indicator, like the thumb of a scroll bar:

- The default method calls the Toolbox routine `TrackControl()` with no action proc.
- If the value of the control has changed (the user moved the indicator to a new place), this method sends a `DoThumbDragged()` message to the control.

```
void DoThumbDragged(short delta);
```

An indicator in a control has been moved. The `DoClick()` method sends a `DoThumbDragged()` message to the control when the user changes the position of a control's indicator.

The default method does nothing. Controls with indicators should override this method. For an example of how to override this method, see CScrollBar.

```
void DoGoodClick(short whichPart);
```

The mouse went down and up in the same part of a control.

The default method does nothing. Subclasses should override this method. See the `DoGoodClick()` method in CButton for an example.

CControl summary

```
struct CControl : CPane {
    ControlHandle   macControl;

    void        Dispose(void);

    void        SetValue(short aValue);
    short       GetValue(void);
    void        SetMaxValue(short aMaxValue);
    short       GetMaxValue(void);
    void        SetMinValue(short aMinValue);
    short       GetMinValue(void);
    void        SetTitle(Str255 aTitle);
    void        GetTitle(Str255 aTitle);
    void        SetActionProc(VoidFunc anActionProc);

    void        Hide(void);
    void        Activate(void);
    void        Deactivate(void);
    void        Offset(short hOffset, short vOffset, Boolean redraw);
    void        ChangeSize(Rect *delta, Boolean redraw);

    void        Draw(Rect *area);
    void        DrawAll(Rect *area);
    void        Prepare(void);

    void        DoClick(Point hitPt, short modifierKeys, long when);
    void        DoThumbDragged(short delta);
    void        DoGoodClick(short whichPart);
};
```

CDataFile 28

Introduction

This class implements the methods you need to work with a Macintosh data file. You can use this class without creating a subclass if you need to read raw bytes. If you require a structured file, you should create a subclass of CDataFile.

Heritage

Superclass	CFile
Subclasses	If you implement a structured data file, you'll want to create a subclass of this class.

Using CDataFile

You can use this class without creating a subclass to read and write the data fork of a Macintosh file. After creating an instance of CDataFile, use one of the specification methods inherited from CFile to indicate which file you're working with. Then use the `Open()` method to open the file, and use the reading and writing methods to read and write the file.

All CDataFile methods return an error code if something goes wrong in a file operation. If nothing goes wrong, they return `noErr`, otherwise they return a Macintosh file system error. For example, if you specify a file that doesn't exist, `Open()` returns an `fnfErr` error code.

Variables

The only instance variable for this class is the refNum of the file. The Macintosh file system uses this number to identify the volume, directory, and file on disk. To learn more about the Macintosh file system, see *Inside Macintosh IV*, Chapter 19, "The File Manager."

`short refNum;`	The file's refNum when open.

Methods

These methods let you open and close data files, read and write data, and get information about the file. You need to use the methods inherited from CFile to specify which file you want to work with.

Construction and destruction methods

```
void  IDataFile(void)
```

Initialize the data file.

Accessing methods

```
OSErr SetLength(long aLength);
```

Set the end-of-file marker for this file at `aLength`.

```
OSErr GetLength(long *theLength);
```

Return the length of the file in `theLength`.

```
OSErr SetMark(long howFar, short fromWhere);
```

Specify where subsequent read/write operations will take place. This position is called "the mark."

`HowFar` specifies how far in bytes from the `fromWhere` parameter to set the mark. Positive values are offsets toward the end of the file. Negative values are offsets toward the beginning of the file. `FromWhere` is one of

fromWhere value	Meaning
`fsFromStart`	from the beginning of the file
`fsFromLEOF`	from the end of the file
`fsFromMark`	from the current position

```
OSErr GetMark(long *markPos);
```

Return the current position of the mark in `markPos`.

Open/Close methods

```
OSErr Open(SignedByte permission);
```

Open the file with the given permission. `Permission` can be one of:

Permission	Meaning
`fsCurPerm`	same as the current permission
`fsRdPerm`	read permission
`fsWrPerm`	write permission
`fsRdWrPerm`	read/write permission
`fsRdWrShPerm`	shared read/write permission

See *Inside Macintosh IV*, Chapter 19, "The File Manager" for a full discussion of permissions.

```
OSErr Close(void);
```

Write out any unwritten data and close the file.

Read/Write methods

`OSErr ReadAll(Handle *contents);`

Read the entire contents of the file into a handle. This method allocates the handle. You must pass in a reference to a handle. For example:

```
Handle theData;

myDataFile->ReadAll(&theData);
```

`OSErr ReadSome(Ptr info, long howMuch);`

Read howMuch bytes into a buffer starting at info. You must have already allocated the space to read the information into. For example:

```
char   myBuffer[128];

myDataFile->ReadSome(myBuffer, 128L);
```

`OSErr WriteAll(Handle contents);`

Write the contents of the handle to the file.

`OSErr WriteSome(Ptr info, long howMuch);`

Write howMuch bytes from the buffer starting at info.

CDataFile summary

```
struct CDataFile : CFile {
    short      refNum;     /* Reference number when open        */

    void       IDataFile(void);

    OSErr      SetLength(long aLength);
    OSErr      GetLength(long *theLength);
    OSErr      SetMark(long howFar, short fromWhere);
    OSErr      GetMark(long *markPos);

    OSErr      Open(SignedByte permission);
    OSErr      Close(void);

    OSErr      ReadAll(Handle *contents);
    OSErr      ReadSome(Ptr info, long howMuch);
    OSErr      WriteAll(Handle contents);
    OSErr      WriteSome(Ptr info, long howMuch);
};
```

CDecorator
29

Introduction

CDecorator implements an object to arrange windows on the screen. Usually there is only one instance of this class.

Heritage

Superclass	CObject
Subclasses	None

Using CDecorator

CDecorator gives you methods for arranging windows on the screen. There is only one instance of CDecorator which is stored in the global variable `gDecorator`.

After creating and initializing a new window, send it in a `PlaceNewWindow()` message to the decorator. The decorator makes the window a bit smaller than the main screen and offsets it down and to the right. Using the decorator is entirely optional.

You can create a subclass of CDecorator if your application requires it. You might want to implement a decorator that implements window tiling, for instance. Be sure to override the `MakeDecorator()` method in your application subclass to initialize your decorator subclass and to store it in the global variable `gDecorator`.

Variables

The global decorator object, created in the application method `MakeDecorator()`, is stored in the global variable `gDecorator`.

Global variable

`CDecorator *gDecorator;`	The window-dressing object.

Instance variables

`short wCount;`	The number of new windows placed
`short index;`	Index for offsetting windows
`short wWidth;`	Width of a window in pixels
`short wHeight;`	Height of a window in pixels
`short hLocation;`	Horizontal location for next window
`short vLocation;`	Vertical location for next window

Methods

`void IDecorator(void);`

Initialize the decorator. The application method `MakeDecorator()` creates the global decorator and sends it this message.

`void PlaceNewWindow(CWindow *theWindow);`

Place `theWindow` on the screen. The default method makes it fill up most of the screen and offsets it down and to the right of the last window the decorator placed.

`void CenterWindow(CWindow *theWindow);`

Center the window specified by `theWindow` on the main screen. This method does not increment `wCount` or change any of the instance variables.

`short GetWCount(void);`

Return the number of windows the decorator has put on the screen. You can use this value to give your untitled windows sequential numbers like Untitled-1.

CDecorator summary

```
struct CDecorator : CObject {
    short       wCount;
    short       index;
    short       wWidth;
    short       wHeight;
    short       hLocation;
    short       vLocation;

    void        IDecorator(void);
    void        PlaceNewWindow(CWindow *theWindow);
    void        CenterWindow(CWindow *theWindow);
    short       GetWCount(void);
};
```

CDesktop
30

Introduction

CDesktop implements a view that occupies the entire screen. This view serves as the top of the visual hierarchy and the enclosure for all windows. Normally, there is only one instance of CDesktop.

Heritage

Superclass	CView
Subclasses	None

Using CDesktop

The most common way you'll use the desktop is as the enclosure for your windows. Your application should not send messages to the desktop. Instead, you should rely on higher level objects like windows and directors to send messages to it.

The `More Classes` folder contains a subclass of CDesktop called CFWDesktop which implements a desktop that supports floating windows.

The desktop is stored in the global variable `gDesktop`. The application method `MakeDesktop()` creates an instance of the CDesktop and stores it in that variable. If you use CFWDesktop, be sure to override `MakeDesktop()`.

Variables

The global variable `gDesktop` holds a pointer to the single instance of the desktop. Use this variable to specify the enclosure for your application's windows.

Global variable

`CDesktop *gDesktop;`	The single instance of the desktop

Instance variables

`Rect bounds;`	Boundaries of the desktop
`CList *itsWindows;`	List of windows
`CWindow *topWindow;`	Top-most application window

Methods

Construction/Destruction methods

```
void IDesktop(CBureaucrat *aSupervisor);
```

Initialize the desktop. The default method opens a GrafPort that your application uses as its desktop. The desktop's supervisor should be the application, stored in the global variable `gApplication`.

```
void Dispose();
```

Dispose of the desktop and send a `Dispose()` message to all of the desktop's windows.

Appearance methods

```
void Show(void);
```

Makes a desktop visible by showing all of its windows. This method sends a `Show()` message to all of the desktop's windows.

```
void Hide(void);
```

Hide a desktop by hiding all of its windows. This method sends a `Hide()` message to all of the desktop's windows.

```
void Activate(void);
```

Activate the desktop by activating the top window. This method sends an `Activate()` message to the desktop's top window.

```
void Deactivate(void);
```

Deactivate a desktop by deactivating its top window. This method sends a `Deactivate()` message to the desktop's top window.

```
Boolean ReallyVisible(void);
```

Return TRUE if the desktop is visible.

Mouse methods

```
void DispatchClick(EventRecord *macEvent);
```

If the user clicks the mouse anywhere, find out where the mouse went down, and send the appropriate message to the object the click is intended for. This method uses what the Macintosh Toolbox routine `FindWindow()` returns to determine what message to send to which object.

Part	Action
`inDesk`	If the mouse goes down in the desktop, this method sends a `DoClick()` message to the desktop. The default `DoClick()` method does nothing.
`inSysWindow`	If the mouse goes down in a system window, this method calls the Toolbox routine `SystemClick()`.
`inMenuBar`	If the mouse goes down in the menu bar, this method calls the Toolbox routine `MenuSelect()`. This method then sends a `FindCmdNumber()` message to the bartender to convert the menu selection into a command number. This command number is sent to the gopher in a `DoCommand()` message.
`inContent`	If the mouse goes down in the content region of a window and the window is not active, this method sends the window a `Select()` message. If the window is already active and can receive clicks,this method sends the window a `DispatchClick()` message which will eventually send a `DoClick()` message to a pane or the window itself.
`inDrag`	If the mouse goes down in the drag region (the title bar), this method sends the window a `Drag()` message.
`inGrow`	If the mouse goes down in the window's grow region (the lower right corner), this method sends the window a `Resize()` message.
`inGoAway`	If the mouse goes down and up in the window's close box, this method sends it a `Close()` message.
`inZoomIn` `inZoomOut`	If the mouse goes down and up in the window's zoom box, this method sends it a `Zoom()` message.

```
void DoMouseUp(EventRecord *macEvent);
```

Handle a mouse up event in the desktop. The default method does nothing.

```
void DispatchCursor(Point where, RgnHandle mouseRgn);
```

Set the cursor for the view the cursor is in. If the cursor is over an active window, this method sends a `DispatchCursor()` message to the window. If the cursor is in the menu bar, this method sets it to the arrow. If the cursor is not in the menu bar or in an active win-

dow, this method sends an `AdjustCursor()` message to the desktop. You should not override this method.

```
void AdjustCursor(Point where, RgnHandle mouseRgn);
```
Adjust the cursor shape when the mouse is in an insignificant part of the desktop. The default method sets the cursor to an arrow.

```
Boolean Contains(Point thePoint);
```
Return TRUE if `thePoint` is within the bounds of the desktop.

```
Boolean HitSamePart(Point pointA, Point pointB);
```
Return TRUE if `pointA` and `pointB` are reasonably close together. "Reasonably close" means that the two points are within `REASONABLY_CLOSE` pixels.

Window methods

You do not have to use any of these methods yourself. Directors, objects which manage the interaction between windows and the desktop, take care of sending most of these messages.

```
void AddWind(CWindow *theWindow);
```
Add `theWindow` to the desktop's window list.

```
void RemoveWind(CWindow *theWindow);
```
Remove `theWindow` from the desktop's window list.

```
void SelectWind(CWindow *theWindow);
```
Select `theWindow` and bring `theWindow` to the front.

```
void ShowWind(CWindow *theWindow);
```
Make `theWindow` visible on the desktop.

```
void HideWind(CWindow *theWindow);
```
Hide `theWindow`.

```
void DragWind(CWindow *theWindow, EventRecord *macEvent);
```
Drag `theWindow` on the desktop.

```
void  UpdateWindows(void);
```
Send an `Update()` message to each of the desktop's windows.

Accessing methods

You might want to use these methods when you need to know the size of the desktop.

```
CWindow* GetTopWindow(void);
```

Get the frontmost window.

```
void GetBounds(Rect *theBounds);
```

Get the bounds of the desktop.

```
void GetAperture(Rect *theAperture);
```

Get the visible portion of the desktop. This method returns the bounds of the desktop since all of the desktop is always visible.

Calibration methods

```
void Prepare(void);
```

Set the port to the desktop port. You should not use or override this method.

CDesktop summary

```
struct CDesktop : CView {
    Rect        bounds;
    CList       *itsWindows;
    CWindow     *topWindow;

    void        IDesktop(CBureaucrat *aSupervisor);
    void        Dispose();

    void        Show(void);
    void        Hide(void);
    void        Activate(void);
    void        Deactivate(void);
    Boolean     ReallyVisible(void);

    void        DispatchClick(EventRecord *macEvent);
    void        DoMouseUp(EventRecord *macEvent);
    void        DispatchCursor(Point where, RgnHandle mouseRgn);
    void        AdjustCursor(Point where, RgnHandle mouseRgn);
    Boolean     Contains(Point thePoint);
    Boolean     HitSamePart(Point pointA, Point pointB);

    void        AddWind(CWindow *theWindow);
    void        RemoveWind(CWindow *theWindow);
    void        SelectWind(CWindow *theWindow);
    void        ShowWind(CWindow *theWindow);
    void        HideWind(CWindow *theWindow);
    void        DragWind(CWindow *theWindow, EventRecord *macEvent);
    void        UpdateWindows(void);

    CWindow*    GetTopWindow(void);
    void        GetBounds(Rect *theBounds);
    void        GetAperture(Rect *theAperture);
    void        Prepare(void);
};
```

CDirector
31

Introduction

CDirector is an abstract class for implementing a window that can handle commands. Directors implement communication between the application and window.

Heritage

Superclass	CBureaucrat
Subclasses	CDocument
	CClipboard

Using CDirector

A director is an abstract class that manages the communication between the application and a window. Any time you want to display a window, it must be related to a director.

In most cases, you'll use the CDocument subclass to display and manipulate data stored in a file in a window. The only time you'll create a subclass of CDirector is when you need a special kind of window like a status window or a tear-off menu.

> **Note:** The class CTearOffMenu in the `More Classes` folder implements a tear-off menu as a subclass of CDirector.

When a window belonging to a director becomes the active window, the gopher points to the bureaucrat specified by the director's `itsGopher` instance variable. When the window becomes inactive, the gopher points to the application.

When the switchboard processes a window-related event (such as an update event), it sends the message to the window which in turn sends the message to the director. When the user chooses a command from the menu, the switchboard sends the `DoCommand()` message to the gopher.

The supervisor of a director must be the application. The supervisor of a window must be a subclass of CDirector.

Variables

Global variable

The global variable `gGopher` points to the current bureaucrat which is usually a descendant of CDirector. When a window becomes active `gGopher` points to the bureaucrat specified in the director's `itsGopher` instance variable. When there are no active directors, `gGopher` points to the application.

`CBureaucrat *gGopher;`	The current bureaucrat.

Instance variables

The instance variable `itsGopher` usually points to the main pane of a window. When the director becomes active, the bureaucrat specified in `itsGopher` becomes the gopher.

`CWindow *itsWindow;`	Window that the director controls
`Boolean active;`	TRUE if the director is active
`CBureaucrat *itsGopher;`	Bureaucrat to make the gopher when director is activated.

Methods

Creation and destruction

```
void IDirector(CBureaucrat *aSupervisor);
```

Initialize the director. The supervisor of a director **must** be the application.

This method adds the director to the application's list of directors. By default, the director has no window and is not active. Your director subclass must create the window for the director. Your director subclass should send a `CDirector::IDirector()` message to itself to initialize the director.

```
void Dispose(void);
```

Dispose of the director. This method sends a `Dispose()` message to the director's window (if it has one) and removes the director from the application's list of directors. Your director subclass must call `inherited::Dispose()` to make sure that the director is disposed of properly.

Command methods

```
void DoCommand(long theCommand);
```

Handle a command. The default method handles this command:

`cmdClose`	Send a `Close(FALSE)` message to the director

Your director class will usually override this method. You should handle your own commands first, then call `inherited::DoCommand()` to get the generic effects.

```
void UpdateMenus(void);
```

If a window is associated with the director, this method enables the **Close** command (`cmdClose`).

Appearance methods

```
void Activate(void);
```

A director is becoming active. If the director's window is not a floating window, sets `gGopher` to `itsGopher`. This method sets `gSleepTime` to 0L to force an idle event.

```
void Deactivate(void);
```

A director is becoming inactive. If the director is the gopher, the director's supervisor (the application) becomes the gopher

```
void Suspend(void);
```

The application is being suspended by MultiFinder. The default method sends the director a `Deactivate()` message if the director was active. The director remains active even though the application is suspended.

```
void Resume(void);
```

The application is being resumed under MultiFinder. The default method sends the director an `Activate()` message.

```
Boolean Close(Boolean quitting);
```

Close a director. The default method ignores the quitting parameter and sends the director a `Dispose()` message and returns TRUE

```
Boolean Quit(void);
```

The application is quitting. The default method sends a `Close(TRUE)` message to the director and returns whatever the `Close()` method returns.

Window methods

```
void CloseWind(CWindow *theWindow);
```

Close the window owned by the director. The default method sends a `Close(FALSE)` message to the director. The director usually gets this message when the user clicks `theWindow`'s close box.

If you want a click in the window's close box to mean something other than closing the director, override this method. For instance, you might want to override this method so it hides the window instead of closing it. See the implementation of CClipboard for an example.

```
void ActivateWind(CWindow *theWindow);
```

The window owned by the director has been activated. You generally won't need to use or override this method in your director subclasses unless you want to do something to

theWindow (and not the director) when it becomes active. To perform specific actions at activate time, override the Activate() method instead.

`void DeactivateWind(CWindow *theWindow);`

The window owned by the director is becoming inactive. You generally won't need to use or override this method in your director subclasses unless you want to do something to theWindow (and not the director) when it becomes deactivated. To perform some specific actions at deactivate time, override the Deactivate() method instead.

`Boolean IsActive(void);`

Returns TRUE if the director is active; FALSE otherwise. You should not override this method.

CDirector summary

```
struct CDirector : CBureaucrat {
    CWindow         *itsWindow;
    Boolean         active;
    CBureaucart     *itsGopher;

    void        IDirector(CBureaucrat *aSupervisor);
    void        Dispose(void);

    void        DoCommand(long theCommand);
    void        UpdateMenus(void);

    void        Activate(void);
    void        Deactivate(void);
    void        Suspend(void);
    void        Resume(void);
    Boolean     Close(Boolean quitting);
    Boolean     Quit(void);
    void        CloseWind(CWindow *theWindow);
    void        ActivateWind(CWindow *theWindow);
    void        DeactivateWind(CWindow *theWindow);
    Boolean     IsActive(void);
};
```

CDocument
32

Introduction

CDocument is the main class for presenting and manipulating information. You can think of a document as the association of a window, a file, and a set of panes.

Your application must create a subclass of CDocument.

Heritage

Superclass	CDirector
Subclasses	You must create a subclass of CDocument

Using CDocument

CDocument is one of the classes you need to override to implement an application in the THINK Class Library. You can think of a document as a file that you view through a window. A better way to think about a document is that it is the essence of a Macintosh application. It is anything that you can display and manipulate inside a window.

The document is where your application draws and displays its data. All documents have windows associated with them. Most documents also have an associated file. Neither the window nor the file are created automatically. You must create them yourself.

Your document class should override these methods:

```
Initialization method          OpenFile()
Dispose()                      DoSave()
DoCommand()                    DoSaveAs()
NewFile()                      Revert()
```

Your document class must have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By convention, the name of your initialization method should be `IYourDoc()` where `YourDoc` is the name of your document class. Your initialization method should call `CDocument::IDocument()`. The supervisor of a document is always `gApplication`.

If your application allocates memory, you should also override the `Dispose()` method to deallocate it. Be sure that your method calls `inherited::Dispose()` to make sure that the document is disposed of properly.

> **Note:** You do not need to dispose of the `itsWindow` or the `itsFile` instance variables. The default `Dispose()` method does that for you.

Your document class's `DoCommand()` method does most of the work in your application. When a window is active, the switchboard will send all commands to the document first (it's the gopher), and if the document can't handle it, the application class tries to handle it. Your document class should handle all the commands it knows about, and call `inherited::DoCommand()` when it can't.

> **Note:** Be sure that your `DoCommand()` method or that one of the methods it invokes sets the instance variable `dirty` to TRUE when there has been a change to the document.

Your document class will get a `NewFile()` message when the user chooses **New** from the **File** menu. This method needs to create a window and attach the panes for it. The `NewFile()` method doesn't need to create a file until the user tries to save the document.

Your document gets an `OpenFile()` message when the user chooses **Open...** from the **File** menu. The `OpenFile()` method has one argument: a pointer to a Macintosh `SFReply` record. When you get the `OpenFile()` message, you can be sure that the `SFReply` record is properly filled in. Your `OpenFile()` message needs to create an instance of a file object (usually of class CDataFile). You can send your file any of several read messages to obtain its contents. Your `OpenFile()` method also needs to create a window to display the contents of the file, just as your `NewFile()` method does.

When the user chooses **Save** from the **File** menu, your document gets a `DoSave()` message. Your `DoSave()` method should write the contents of its file to disk. The file object is stored in the instance variable `itsFile`.

When the user chooses **Save As...** from the **File** menu, your document gets a `DoSaveAs()` message. This method takes an `SFReply` record, and you can be sure that it is properly filled in. Your document class needs to override this method to write its data to a file.

If your application supports the **Revert** command, you should implement it in the `DoRevert()` method. Your implementation might do the same thing as closing without saving, then opening the file again.

Variables

Global variables

The global variable gGopher points to the current bureaucrat which is usually a document. When a window becomes active gGopher points to the document that owns it. When there are no active documents, gGopher points to the application.

The supervisor of every document is the application.

`CBureaucrat *gGopher;`	The current bureaucrat.
`CApplication *gApplication;`	The application

Instance variables

`CPane *itsMainPane;`	The document's main pane. NULL if the document has no main pane. The main pane's enclosure should be `itsWindow`, an instance variable inherited from CDirector.
`CFile *itsFile;`	The file associated with this document. NULL if document has no file.
`CTask *lastTask;`	The last task this document was notified as being completed
`Boolean undone;`	TRUE if the last task was undone.
`CPrinter *itsPrinter;`	The printer object associated with this document. NULL if document is not printable
`Boolean dirty;`	TRUE if document has been altered.
`short pageWidth;`	The width of a page.
`short pageHeight;`	The hight of a page.

Methods

Construction and destruction methods

```
void IDocument(CBureaucrat *aSupervisor, Boolean printable);
```

Initialize the document. `ASupervisor` must be the `gApplication`. If the value of `printable` is TRUE, this method creates an instance of CPrinter and stores it in the `itsPrinter` instance variable.

```
void Dispose(void);
```

Dispose of the document and memory it allocated. The `Close()` method usually sends the document a `Dispose()` message. If your document subclass allocates memory (such as a pane for `itsMainPane`) your `Dispose()` method must dispose of it, then you must invoke `inherited::Dispose()` to dispose of the rest of the document.

Command methods

```
void Notify(CTask *theTask);
```

A subordinate has completed a task. The default method disposes the current `lastTask` and stores `theTask` in the instance variable `lastTask`. Then, this method sends an `UpdateUndo()` message to the document. It also enables the `cmdSave`, `cmdSaveAs`, and `cmdRevert` commands according to the value of the document's `dirty` flag.

```
void DoCommand(long theCommand);
```

Handle a document-related command. The default method handles these commands:

Command	Action
`cmdClose`	Send a `Close()` message to the document.
`cmdSave`	Set the cursor to the watch cursor and send a `DoSave()` message to the document.
`cmdSaveAs`	Send a `DoSaveFileAs()` message to the document.
`cmdRevert`	Display a "Do you really want to revert alert." If the user responds OK, set the cursor to the watch cursor and send a `DoRevert()` message to the document.
`cmdPageSetup`	If the document is printable, send a `DoPageSetup()` to the document.
`cmdPrint`	If the document is printable, send a `DoPrint()` message to the document.
`cmdUndo`	If there is a last task, send either a `Redo()` or an `Undo()` message to the task. Then send an `UpdateUndo()` message to the document.

Your document class will usually override this method. You should handle your own commands first, then call `inherited::DoCommand()` to handle the default commands.

```
void UpdateMenus(void);
```

Update the menu items right before they appear on the screen. The default method enables the following commands:

Command	Enabled if...
`cmdSaveAs`	always
`cmdSave`	the document is dirty.
`cmdRevert`	a file is associated with the document and it's dirty.
`cmdPageSetup`	the document is printable.
`cmdPrint`	the document is printable.
`cmdUndo`	there is a last task to undo.

Your document class should override this method to enable the appropriate commands for your document. Be sure you call `inherited::UpdateMenus()` **before** you enable your document's commands.

Appearance Methods

```
Boolean Close(Boolean quitting);
```

A document is being closed. This method sends a `ConfirmClose()` message to the document. If the result is TRUE it sends a `Close()` message to `itsFile` (if it has one). The `quitting` parameter tells the `ConfirmClose()` method whether to ask to save before "closing" or "quitting." If the document was actually closed, this method returns TRUE. Otherwise it returns FALSE.

```
void CloseWind(CWindow *theWindow);
```

Close the specified window. The default method sends a `Close()` message to the document. If you want to be able to close windows without actually closing a file, you should override this method. Otherwise, your document class should not override this method.

```
Boolean ConfirmClose(Boolean quitting);
```

Display a "Save before closing?" or "Save before quitting?" dialog. If the user answers yes, send a `DoSave()` to the document and return TRUE. If the user answers no, just return TRUE. If the user answers Cancel, return FALSE. If your document does not have a file, your document class should override this method with a method that always returns TRUE.

File Creation

```
void NewFile(void);
```

Open a new file. The default method does nothing. Your `CreateDocument()` method in your application class should send a `NewFile()` message to the document it creates. Your document class should override this method to do the following:

- Create a new window and assign it to `itsWindow`
- Create a new file and assign it to `itsFile`
- Create the panes you need and assign the main pane to `itsMainPane`

```
void OpenFile(SFReply *macSFReply);
```

Open an existing file. The default method does nothing. Your `OpenDocument()` method in your application class should send an `OpenFile()` message to the document it creates. Your document class should override this method and do the following:

- Create a new window and assign it to `itsWindow`
- Open the file specified in the `macSFReply` and assign it to `itsFile`
- Create the panes you need and assign the main pane to `itsMainPane`
- Display the contents of the file in the pane

> **Note:** To learn how to specify a file, see the class CFile. To learn how to read and write from a data file, see the class CDataFile.

Printing Methods

```
short PageCount(void);
```

Return the number of pages in the document. The default method returns the number of pages based on the pixel extent and the size of the page. See the method `GetPixelExtent()` in the CPane and CPanorama classes.

> **Note:** If the document doesn't have a CPrinter object associated with it, it uses the constants `STD_PAGE_WIDTH` and `STD_PAGE_HEIGHT` for the size of the page.

```
void AboutToPrint(short *firstPage, short *lastPage);
```

Check the range of pages to be printed. The default method changes `lastPage` to be equal to `PageCount()`. Your document class should override this method to do whatever is appropriate for your application. This is the place where the document can request information about the page size from `itsPrinter`.

```
void PrintPageOfDoc(short pageNum);
```

Print the specified page. The default method sends a `PrintPage()` message to `itsMainPane` if it's not NULL. This method must not send messages to `itsPrinter`.

```
void DonePrinting(void)
```

Printing is complete. The `PrintPageRange()` method in CPrinter sends this message when the print loop is over. This method sends a `DonePrinting()` message to `itsMainPane`.

Filing Methods

```
Boolean DoSave(void);
```

Save the document under its current name and return TRUE if successful. The current file is available through the instance variable `itsFile`. The default method does nothing. Your document class must override this method.

```
Boolean DoSaveAs(SFReply *macSFReply);
```

Save the document under a new name and return TRUE if successful. The `macSFReply` record specifies where to write the file. The default method does nothing. Your document class must override this method.

```
void DoRevert(void);
```

Revert to the last saved version of this document. The default method does nothing. If you want your application to support the **Revert** command, you'll have to override this method.

```
Boolean DoSaveFileAs(void);
```

Respond to a **Save As...** command and return TRUE if successful. This method sends a `PickFileName()` message to the document. If the user provides a good file name, it sends a `DoSaveAs()` message to the document. Since this method implements the standard **Save As...** command through two other messages, you won't need to override this method.

```
void PickFileName(SFReply *macSFReply);
```

Display a standard get file dialog to let the user choose a new name. It uses the `GetName()` method to specify the default name.

```
void GetName(Str255 theName);
```

Get the name of the document. If there is a file associated with the document, this method returns the name of the file. If there is no file associated with the document, but there is a window associated with it, it returns the the title of the window. If there is neither a file nor a window associated with the document, this method returns a null string.

Undo methods

```
void UpdateUndo();
```

Update the Undo/Redo menu item to have the correct wording. Default method sends a `GetNameIndex()` message to the `lastTask` to find its string in the `STRtaskNames` `STR#` resource. Your document class should not override this method.

Class resources

`STRprompt 150`	STR resource ID for PickFileName() prompt string
`ALRTrevert 150`	Revert to saved alert
`ALRTsaveChanges 151`	Save changes before close/quit alert
`STRtaskNames 130`	STR# resource ID for task names
`STRcommon 128`	STR# resource for commonly used strings

CDocument summary

```
struct CDocument : CDirector {
    CPane                *itsMainPane;
    CFile                *itsFile;
    CTask                *lastTask;
    Boolean              undone;
    struct CPrinter      *itsPrinter;
    Boolean              dirty;
    short                pageWidth;
    short                pageHeight;

    void       IDocument(CBureaucrat *aSupervisor, Boolean printable);
    void       Dispose(void);

    void       Notify(CTask *theTask);
    void       DoCommand(long theCommand);
    void       UpdateMenus(void);

    Boolean    Close(Boolean quitting);
    void       CloseWind(CWindow *theWindow);
    Boolean    ConfirmClose(Boolean quitting);

    void       NewFile(void);
    void       OpenFile(SFReply *macSFReply);

    short      PageCount(void);
    void       AboutToPrint(short *firstPage, short *lastPage);
    void       PrintPageOfDoc(short pageNum);
    void       DonePrinting(void);

    Boolean    DoSave(void);
    Boolean    DoSaveAs(SFReply *macSFReply);
    void       DoRevert(void);
    Boolean    DoSaveFileAs(void);
    void       PickFileName(SFReply *macSFReply);
    void       GetName(Str255 theName);

    void       UpdateUndo(void);
};
```

CEditText
33

Introduction

CEditText implements a pane that displays editable text. This class uses the Macintosh Text Edit routines.

Heritage

Superclass	CStaticText
Subclasses	None

Using CEditText

Use CEditText whenever you need to display and edit text. An edit text pane is usually the panorama in a scroll pane so you can scroll through the text. The `DoCommand()` method of an edit text pane handles all the common text editing commands such as cutting and pasting, font selection, line spacing, etc.

To make sure that the edit text pane responds to commands, you must place it in the chain of command. The best way to do this is to set the value of your document's `itsGopher` instance variable to the edit pane.

Because CEditText is based on the Macintosh Text Edit (TE) routines, it has some limitations. You should not use CEditText to implement text editors. It's designed to edit small amounts of text. The maximum number of characters you can store in a CEditText record is around 32,000, but you'll notice performance degradation long before it gets that big.

> **Note:** CEditText does not use the Styled Text Edit routines described in *Inside Macintosh V*.

Variables

`short fontItem;`	Menu item number of the current font.
`short sizeItem;`	Menu item number of the current size.

Methods

Construction and destruction methods

```
void IEditText(CView *anEnclosure, CBureaucrat *aSupervisor, short
aWidth, short aHeight, short aHEncl, short aVEncl, SizingOption
aHSizing, SizingOption aVSizing, short aLineWidth);
```

Initialize an edit text pane. Most of the arguments to this method are identical to the pane initialization. `ALineWidth` specifies how wide the lines should be. If it's less than zero, the width is the same as the Macintosh TE record's `viewRect`.

> **Note:** The descriptions of the other arguments are in CPane.

```
void IViewRes(ResType rType, short resID, CView *anEnclosure,
CBureaucrat *aSupervisor);
```

Initialize edit text pane from a resource template. `Rttype` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the same as for `IBorder()`. This method is inherited from CView.

To initialize an edit text pane from a resource file, use a `'StTx'` resource.

Command methods

```
void DoCommand(long theCommand);
```

The edit text pane's `DoCommand()` method handles the most common commands that apply to text: cutting and pasting, font selection, styling, alignment, and spacing.

The default method handles these commands:

- Font selection from the **Font** menu (`MENUfont`)
- Size selection from the **Size** menu (`MENUsize`)
- Editing commands: `cmdCut`, `cmdCopy`, `cmdPaste`, `cmdClear`
- Style commands: `cmdPlain`, `cmdBold`, `cmdItalic`, `cmdUnderline`, `cmdOutline`, `cmdShadow`, `cmdCondense`, `cmdExtend`
- Alignment commands: `cmdAlignLeft`, `cmdAlignCenter`, `cmdAlignRight`
- Spacing commands: `cmdSingleSpace`, `cmd1HalfSpace`, `cmdDoubleSpace`

```
void UpdateMenus(void);
```

Update the menus that have to do with text processing right before the menu appears. This method enables the **Cut**, **Copy**, and **Clear** commands if there is a current selection and enables the **Paste** command if there is text in the clipboard. This menu also checks the current font, size, style, alignment, and spacing commands.

> **Note:** See the implementation of this method if you want an example of how to write your own `UpdateMenus()` method.

Mouse and Keystrokes methods

```
void DoClick(Point hitPt, short modifierKeys, long when);
```

Handle a mouse down in an edit text pane.

```
void DoKeyDown(char theChar, Byte keyCode, EventRecord *macEvent);
```

Handle a key down in an edit text pane. Inserts `theChar` into the text.

```
void DoAutoKey(char theChar, Byte keyCode, EventRecord *macEvent);
```

Handle an auto-key in an edit text pane. Inserts `theChar` into the text. If the command key is down, this method does nothing.

Display methods

```
void Activate(void);
```

Activate the edit text pane.

```
void Deactivate(void);
```

Deactivate the edit text pane.

Printing methods

```
void AboutToPrint(short *firstPage, short *lastPage);
```

The specified range of pages is about to be printed. This method deactivates the text edit record to unhighlight the current selection.

```
void DonePrinting(void);
```

Printing is over. This method re-highlights the current selection if the edit pane is active.

Cursor methods

```
void Dawdle(long *maxSleep);
```

This method flashes the insertion point when the edit text pane is active. CEditText's `Dawdle()` method sets `maxSleep` to the value of `GetCaretTime()`, which is the rate at which the insertion point blinks. Setting this value ensures that `WaitNextEvent()` will generate a null event at least that often.

> **Note:** To learn more about "sleep time," see the description of the `Dawdle()` message in CBureaucrat.

```
void AdjustCursor(Point where, RgnHandle mouseRgn);
```

Turn the cursor into an I-beam when the mouse is in the edit text pane.

CEditText summary

```
struct CEditText : CStaticText {
    short       fontItem;
    short       sizeItem;

    void        IEditText(CView *anEnclosure, CBureaucrat *aSupervisor,
                      short aWidth, short aHeight,
                      short aHEncl, short aVEncl,
                      SizingOption aHSizing, SizingOption aVSizing,
                      short aLineWidth);

    void        DoCommand(long theCommand);
    void        UpdateMenus(void);

    void        DoClick(Point hitPt, short modifierKeys, long when);
    void        DoKeyDown(char theChar, Byte keyCode, EventRecord *macEvent);
    void        DoAutoKey(char theChar, Byte keyCode, EventRecord *macEvent);

    void        Activate(void);
    void        Deactivate(void);

    void        AboutToPrint(short *firstPage, short *lastPage);
    void        DonePrinting(void);

    void        Dawdle(long *maxSleep);
    void        AdjustCursor(Point where, RgnHandle mouseRgn);
};
```

CEnvironment
34

Introduction

CEnvironment maintains a drawing environment for any pane.

Heritage

Superclass	CObject
Subclasses	None

Using CEnvironment

Every pane has an `itsEnvironment` instance variable. If this variable points to an object of this class, the `Prepare()` method will send it a `Restore()` message to set up the QuickDraw drawing environment for the pane.

You can use this class to make sure that the drawing environment is set up correctly. Or you might want to change the drawing environment according to some saved settings.

Variables

This class has no instance variables.

Methods

`void Restore();`
Restore the drawing environment in the current port. The default method just calls the QuickDraw routine `PenNormal()`.

CEnvironment summary

```
struct CEnvironment : CObject {
    void        Restore(void);
};
```

CError
35

Introduction

CError implements a set of error handling methods. There is a global error handler you can use to report errors.

Heritage

Superclass	CObject
Subclasses	None

Using CError

The application initialization method `IApplication()` creates an instance of CError and stores it in the global variable `gError`. Several objects in the THINK Class Library use the global error handler to post error messages.

The default error handler just reports messages and gives you a chance to quit or to proceed with the application. You can create a subclass of this method to implement more sophisticated error recovery.

Variables

Global variable

`CError *gError;`	Global error handler

Instance variable

This class has no instance variables.

Methods

Error reporting methods

```
void SevereMacError(OSErr macErr);
```

Report an operating system error in a Stop Alert that gives the user a chance to quit the application or jump back to the beginning of the event loop. This method looks for an `Estr` resource with the same ID as a Macintosh operating system error. If it doesn't find one, it uses the default string (`STR 200`).

If the user presses the Quit button, this method sends a `Quit()` message to the application. If the user presses the Proceed button, this method sends the application a `JumpToEventLoop()` message.

```
Boolean CheckOSError(OSErr macErr);
```

Check for a Macintosh operating system error. Return TRUE if everything is OK, FALSE otherwise. If `macErr` is an error, this method displays a Stop Alert. This method looks for an `Estr` resource with the same ID as the operating system error. If it can't find one, it uses a default string.

```
void PostAlert(short STRid, short index);
```

Post a general alert and return. `STRid` is the resource ID of a `STR#` resource. Index is the index into the `STR#` resource. The string is displayed in a generic alert box.

Error handling functions

Note that these are functions, not methods.

```
pascal long GrowZoneFunc(Size bytesNeeded);
```

The application method `InitMemory()` installs this function as the application's GrowZone function which is called in low memory conditions. This function sends a `GrowMemory()` message to the application. See CApplication.

```
void CheckResource(Handle h);
```

If the handle `h` is NULL, this method sends a `SevereMacError(resNotFound)` message to the global `gError`. You should call this function after trying to retrieve a resource.

```
void CheckAllocation(void *p);
```

If `p` is a NULL, the method sends a `SevereMacError(MemError())` to report a severe memory error. You should call this function after trying to allocate memory.

Class resources

`ALRT/DITL 128`	Generic alert box. Contains `^0` for use with `ParamText()`.
`ALRT/DITL 200`	Alert box for a `SevereError()`
`ALRT/DITL 300`	Alert box for a Macintosh operating system error.
`STR 300`	String that reports a severe Macintosh system error
`Estr` resources	An `Estr` resource has the same format as a `STR` resource. The `SevereMacError()` method looks for an `Estr` resource with the same ID as an operating system error. For instance, `Estr -42` might read "Too many files open."

CError summary

```
struct CError : CObject {
    void      SevereMacError(OSErr macErr);
    Boolean   CheckOSError(OSErr macErr);
    void      PostAlert(short STRid, short index);
};

pascal long   GrowZoneFunc(Size bytesNeeded);
void    CheckResource(Handle r);
void    CheckAllocation(void *p);
```

CFile
36

Introduction

CFile is an abstract class for implementing classes that deal with disk files.

Heritage

Superclass	CObject
Subclasses	CDataFile

Using CFile

CFile is an abstract class for dealing with Macintosh disk files. Most of the time you'll use the CDataFile subclass to work with regular data files.

Before you open a file, you must specify it. Specifying means identifying it to the Macintosh file manager. This class gives you three ways to specify a file depending on the type of information you can provide. The most common specification method, `SFSpecify()`, lets you use an `SFReply` record from the standard file dialogs to identify a file.

Some CFile methods return an error code if something goes wrong in a file operation. If nothing goes wrong, they return `noErr`, otherwise they return a Macintosh file system error. For example, if you specify a file that doesn't exist, `Open()` returns an `fnfErr` error code.

Variables

`Str63 name;`	File name
`short volNum;`	Volume containing the file
`long dirID;`	Directory within the volume

Methods

Construction and destruction methods

```
void IFile(void);
```

Initialize the file object.

```
void Dispose(void);
```
Close and dispose of the file object.

Specifying methods

```
void Specify(Str63 aName, short aVolNum);
```
Specify a file by its name and volume reference number. Use this method to specify files on MFS volumes.

```
void SpecifyHFS(Str63 aName, short aVolNum, long aDirID);
```
Specify a file by its name, volume number, and directory ID.

```
void SFSpecify(SFReply *macSFReply);
```
Specify a file from the information in a macSFReply record. Use this method to specify a file that the user chose through a standard file dialog.

Open and close methods

```
OSErr Open(SignedByte permission);
```
Open the file with the specified permission. This method does nothing. Subclasses must override this method. This method returns `noErr` if the operation was successful and a Macintosh OS error code if it was not.

```
OSErr Close(void);
```
Close this file. This method does nothing. Subclasses must override this method. Your method returns `noErr` if the operation was successful and a Macintosh OS error code if it was not.

Accessing methods

```
void GetName(Str63 theName);
```
Get the name of the file.

Filing methods

```
OSErr CreateNew(OSType creator, OSType fType);
```
Create new file with the specified creator and file type. This method uses the name and volume information you set up with one of the specification methods. You can use the application signature in `gSignature` for the creator. This method returns `noErr` if the operation was successful and a Macintosh OS error code if it was not.

```
OSErr ThrowOut(void);
```
Close the file and delete it. This method returns `noErr` if the operation was successful and a Macintosh OS error code if it was not.

`OSErr ChangeName(Str63 newName);`

Give this file a new name. This method returns `noErr` if the operation was successful and a Macintosh OS error code if it was not.

CFile summary

```
struct CFile : CObject {

    Str63     name;
    short     volNum;
    long      dirID;


    void      IFile(void);
    void      Dispose(void);

    void      Specify(Str63 aName, short aVolNum);
    void      SpecifyHFS(Str63 aName, short aVolNum, long aDirID);
    void      SFSpecify(SFReply *macSFReply);

    OSErr     Open(SignedByte permission);
    OSErr     Close(void);

    void      GetName(Str63 theName);

    OSErr     CreateNew(OSType creator, OSType fType);
    OSErr     ThrowOut(void);
    OSErr     ChangeName(Str63 newName);
};
```

CList
37

Introduction

CList implements an ordered list of objects.

Heritage

Superclass	CCluster
Subclasses	None

Using CList

Use an object of class CList when you need to maintain an ordered list of objects. Several of the objects in the THINK Class Library use CList to maintain lists. You can use the iteration methods inherited from CCluster to apply functions to each item in a list. Every object in a list has an index. Index values begin at 1.

Variables

This class has no instance variables.

Methods

For methods that insert new objects before or after existing objects, be sure that the existing object is actually in the list. If you don't, strange things may happen.

Construction methods

CList inherits the dispose method from CCluster.

```
void IList(void);
```

Initialize the list.

Insertion and deletion methods

```
void Remove(CObject *theObject);
```

Remove `theObject` from the list. All following objects move up one slot.

```
void Append(CObject *theObject);
```
Add `theObject` to the end of the list.

```
void Prepend(CObject *theObject);
```
Add `theObject` to the beginning of the list.

```
void InsertAfter(CObject *theObject, CObject *afterObject);
```
Insert the `theObject` after the object specified by `afterObject`.

```
void InsertAt(CObject *theObject, long index);
```
Make theObject be the indexth object in the list. All subsequent objects move down one position

Ordering methods

```
void BringFront(CObject *theObject);
```
Make `theObject` be the first object in the list.

```
void SendBack(CObject *theObject);
```
Make `theObject` be the last object in the list.

```
void MoveUp(CObject *theObject);
```
Move `theObject` up one slot toward the front of the list. The object that was before it in the list moves down one slot.

```
void MoveDown(CObject *theObject);
```
Move `theObject` down one slot toward the end of the list. The object that was after it in the list moves up one slot.

```
void MoveToIndex(CObject *theObject, long index);
```
Move `theObject` to the `index`th position in the list. If `theObject` moves back in the list, the items between its original position and its new position move up one slot. If `theObject` moves forward in the list, the items between its new position and its original position move down one slot.

Membership methods

```
CObject* FirstItem(void);
```
Return the first item in the list.

```
CObject* LastItem(void);
```
Return the last item in the list.

```
CObject* NthItem(long n);
```
Return the nth item in the list.

```
long FindIndex(CObject *theObject);
```
Return the index of the object. Indexes begin at 1. If the item is not in the list, this method returns 0.

```
CObject* FirstSuccess(BooleanFunc testFunc);
```
Return the first item that causes the `testFunc` function to return TRUE. `TestFunc` must be declared like this:

```
Boolean MytestFunc(CObject *theObject);
```

```
CObject* FirstSuccess1(BooleanFunc testFunc, long param);
```
Return the first item that causes the `testFunc` function to return TRUE. `TestFunc` must be declared like this:

```
Boolean MytestFunc(CObject *theObject, long param);
```

```
CObject* LastSuccess(BooleanFunc testFunc);
```
Return the last item in the list that causes the `testFunc` function to return TRUE. `TestFunc` must be declared like this:

```
Boolean MytestFunc(CObject *theObject);
```

```
CObject* LastSuccess1(BooleanFunc testFunc, long param);
```
Return the last item in the list that causes the `testFunc` function to return TRUE. `TestFunc` must be declared like this:

```
Boolean MytestFunc(CObject *theObject, long param);
```

CList summary

```
struct CList : CCluster {

    void        IList(void);

    void        Remove(CObject *theObject);
    void        Append(CObject *theObject);
    void        Prepend(CObject *theObject);
    void        InsertAfter(CObject *theObject, CObject *afterObject);
    void        InsertAt(CObject *theObject, long index);

    void        BringFront(CObject *theObject);
    void        SendBack(CObject *theObject);
    void        MoveUp(CObject *theObject);
    void        MoveDown(CObject *theObject);
    void        MoveToIndex(CObject *theObject, long index);

    CObject*    FirstItem(void);
    CObject*    LastItem(void);
    CObject*    NthItem(long n);
    long        FindIndex(CObject *theObject);
    CObject*    FirstSuccess(BooleanFunc testFunc);
    CObject*    FirstSuccess1(BooleanFunc testFunc, long param);
    CObject*    LastSuccess(BooleanFunc testFunc);
    CObject*    LastSuccess1(BooleanFunc testFunc, long param);
};
```

CMouseTask
38

Introduction

CMouseTask is an abstract class that implements mouse tracking.

Heritage

Superclass	CTask
Subclasses	You must define a subclass of this class.

Using CMouseTask

CMouseTask is an abstract class that lets you implement undoable mouse-related actions. For example, if you're writing a drawing application, you want to make sure that you can undo anything that you move or draw.

You don't have to use this class to implement mouse-related actions. You can always track the mouse yourself in your pane's `DoClick()` method. If you do use CMouseTask to implement mouse actions, you don't have to make them undoable.

Defining a mouse task

To implement a mouse tracking task, define a subclass of CMouseTask and override the `KeepTracking()` and `EndTracking()` methods. The `KeepTracking()` method does whatever you want to happen while the mouse is down. The `EndTracking()` method does whatever you want to happen when the mouse is released.

For example, if you're moving a rectangle from one place in a pane to another, the `KeepTracking()` method might draw a gray outline that moves as you move the mouse. The `EndTracking()` method would erase the rectangle from its old location and redraw it in the new location.

If you want to make your mouse task undoable, you need to store enough information in the object to undo the effects of mouse tracking. You must also override the `Undo()` method (inherited from CTask) to use this information to undo the effects of the mouse task.

Using the moving rectangle example again, your `EndTracking()` method might keep the old location of the rectangle in an instance variable. The `Undo()` method would erase the rectangle from its current location and redraw it again in the old location.

Using the mouse task

You use mouse tasks in the `DoClick()` method of a pane. Create an instance of your mouse task, initialize it, and pass it in a `TrackMouse()` message to your pane. This is what part of your `DoClick()` method might look like:

```
CShapeMover *myMover;

myMover= new(CShapeMover);
myMover->IShapeMover(UNDOMoverStr);

this->TrackMouse(myMover, hitPt, &pinRect);
itsSupervisor->Notify(myMover);
```

The `TrackMouse()` method sends your mouse task a `BeginTracking()` message to give you an opportunity to adjust the starting point. As long as the mouse button is down, `TrackMouse()` repeatedly sends `KeepTracking()` messages to your mouse task. When the mouse button is released, `TrackMouse()` sends your mouse task an `EndTracking()` message.

The value you pass to your intialization method is the index of a string in the `STR# 130` resource that describes your task. The document method `UpdateUndo()` uses this string for the wording of the **Undo** command in the **Edit** menu. If your task is not undoable, you can use any value and ignore it.

After you've tracked the mouse, you can send the task in a `Notify()` message to the document. The document will store the task in the document's lastTask instance variable. When you choose **Undo** from the **Edit** menu, the document sends an `Undo()` message to the task to undo the effects of mouse tracking.

Variables

This class has no instance variables

Methods

Construction/Destruction methods

`void IMouseTask(short aNameIndex);`

Initialize a mouse tracking task. `ANameIndex` is the index for the undo/redo string in the `STR# 130` resource.

Mouse tracking methods

```
void BeginTracking(Point *startPt);
```

Mouse tracking is beginning. The document's `TrackMouse()` method sends this message at the beginning of mouse tracking. `StartPt` is a pointer to the starting point (in local coordinates). If your mouse task subclass doesn't alter the starting point, you don't need to override this method.

```
void KeepTracking(Point *currPt, Point *prevPt, Point *startPt);
```

Mouse tracking is under way. The document's `TrackMouse()` method sends this message repeatedly as long as the mouse button is down. `CurrPt` is a pointer to the current mouse location. `PrevPt` is a pointer to the previous mouse location. `StartPt` is a pointer to the original mouse location. Your mouse task subclass must override this method.

```
void EndTracking(Point *currPt, Point *prevPt, Point *startPt);
```

Mouse tracking is over. The document's `TrackMouse()` method sends this message when the mouse button is released. `CurrPt` is a pointer to the current mouse location. `PrevPt` is a pointer to the previous mouse location. `StartPt` is a pointer to the original mouse location. Your mouse task subclass must override this method. If your mouse task is undoable, you should store all the information you need to undo it in this method.

> **Note:** If you're implementing an undoable mouse task, you'll need to override the `Undo()` method as well.

Class resources

`STR# 130`	List of strings that describe undoable tasks. For example, if you're implementing a mouse task that moves graphic images, the string for that task might be "move." The item in the Edit menu would then read "Undo move" or "Redo move."

CMouseTask summary

```
struct CMouseTask : CTask {
    void      IMouseTask(short aNameIndex);

    void      BeginTracking(Point *startPt);
    void      KeepTracking(Point *currPt, Point *prevPt, Point *startPt);
    void      EndTracking(Point *currPt, Point *prevPt, Point *startPt);
};
```

CObject
39

Introduction

CObject is the abstract root level class. It is the ancestor of all the objects in the THINK Class Library.

Heritage

Superclass	None
Subclasses	CBartender
	CBureaucrat
	CChore
	CCollection
	CDecorator
	CEnvironment
	CError
	CFile
	CPrinter
	CSwitchboard
	CTask

Using CObject

Every class in the THINK Class Library is a descendant of this class. CObject is the only class with no superclass. If you need to create a new class which cannot be a subclass of any other class make it a subclass of CObject.

Variables

This class has no instance variables.

Methods

All objects inherit these two methods.

Creation and destruction

```
void Dispose(void);
```

Dispose of the memory an object occupies. If your subclass allocates memory in its initialization method, be sure you release it in its `Dispose()` method.

```
CObject* Copy(void);
```

Make a copy of the object. Note that if a class has instance variables that point to a block of memory, only the reference is copied, not the contents of the block of memory.

CObject summary

```
struct CObject : indirect {
    void        Dispose(void);
    CObject     *Copy(void);
};
```

CPane
40

Introduction

CPane is an abstract class which defines a drawing area within a window or within another pane. In the THINK Class Library, all drawing takes place within a pane. Each pane has its own drawing environment and coordinate system.

Heritage

Superclass	CView
Subclasses	CBorder
	CControl
	CPanorama
	CScrollPane
	CSizeBox

Using CPane

Use a pane when you need a non-scrolling area to draw in within a window. If you need a scrollable area, use CPanorama and CScrollPane. If you need a pane for displaying text, see CStaticText or CEditText.

Your application must define a subclass of CPane (or one of its descendants) to draw in a window. In your subclass, you need to override these methods:

```
IYourPane                    Dispose
Draw                         DoClick
```

Your pane class should have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By convention, the name of your initialization method should be `IYourPane()` where `YourPane` is the name of your pane class. Your initialization method should call `CPane::IPane()` (or whatever pane class you're overriding). The supervisor of a pane should be either the pane that encloses it or the director its window belongs to.

The pane initialization method is where you set the pane's location in its enclosure and its characteristics. If you want your pane to receive clicks, be sure to send the pane a `SetWantsClicks(TRUE)` message, otherwise mouse clicks in your pane are ignored.

If your pane allocates memory, you should also override the `Dispose()` method to deallocate it. Be sure that your method calls `inherited::Dispose()` to make sure that the pane is disposed of properly.

The `Draw()` message tells your pane to draw its contents. You can assume that the port, clip region, and coordinate system have been set up correctly. See "Working with Panes" in the Chapter 16 to learn all about drawing in a pane.

When the user clicks in your pane, it will get a `DoClick()` message. Your `DoClick()` method can either handle the mouse click itself, or it can create a task and send it in a `TrackMouse()` message. To learn more about mouse tracking in a pane, see CMouseTask.

A pane has its own drawing environment. The rectangle that describes the edges of a pane is the pane's **frame**. The frame defines a QuickDraw drawing environment for the pane. In most cases, the top, left corner of the pane is the point (0, 0). The pane's `Prepare()` method takes care of setting up the QuickDraw port and coordinate system, so you don't have to worry about where the pane actually is. The THINK Class Library sends the pane a `Prepare()` message automatically before sending a `Draw()` message and before a `DoClick()` message so the coordinate system is always set up correctly.

Variables

`short width;`	Horizontal size in pixels
`short height;`	Vertical size in pixels
`short hEncl;`	Horizontal location in enclosure
`short vEncl;`	Vertical location in enclosure
`SizingOption hSizing;`	Horizontal sizing option
`SizingOption vSizing;`	Vertical sizing option
`Boolean autoRefresh;`	Refresh after a resize?
`Rect frame;`	Area for displaying the pane which defines the frame coords
`Rect aperture;`	Active drawing area of the pane
`long hOrigin;`	Window left in frame coords
`long vOrigin;`	Window top in frame coords
`CEnvironment *itsEnvironment;`	Drawing environment
`ClipOption printClip;`	The region to clip to when printing.
`Boolean printing;`	Is printing in progress?

Methods

Construction/Destruction methods

```
void IPane(CView *anEnclosure, CBureaucrat *aSupervisor, short
aWidth, short aHeight, short aHEncl, short aVEncl, SizingOption
aHSizing, SizingOption aVSizing);
```

Initialize a pane. Almost all of the descendants of CPane use the same arguments in their initialization methods.

`AnEnclosure` is the enclosing view that contains this pane. Typically the enclosure is either a window or another pane.

`ASupervisor` is the bureaucrat that will handle all the commands that this pane won't. Typically, the supervisor is the document (or director) associated with this pane's window.

`AWidth` and `aHeight` are the width and height of the pane in pixels. `AHEncl` and `aVEncl` are the horizontal and vertical position of the pane within its enclosure.

The `aHSizing` and `aVSizing` parameters specify what happens to the pane when the size of its enclosure changes. The length and height of a pane changes relative to its original position in the enclosing pane. These are the values that `aHSizing` can have:

aHSizing value	Meaning
`sizFIXEDLEFT`	The left edge of the pane is always the same number of pixels from the left edge of the enclosing pane as when it was originally placed.
`sizFIXEDRIGHT`	The right edge of the pane is always the same number of pixels from the right edge of the enclosing pane as when it was originally placed.
`sizFIXEDSTICKY`	The left and right edges stick to their original locations in the enclosing pane. If the enclosure scrolls horizontally, the pane will scroll with it.
`sizELASTIC`	The width of the pane grows and shrinks by the same amount as the enclosing pane.

These are the values the `aVSizing` can have:

aVSizing value	Meaning
`sizFIXEDTOP`	The top edge of the pane is always the same number of pixels from the top edge of the enclosing pane as when it was originally placed.
`sizFIXEDBOTTOM`	The bottom edge of the pane is always the same number of pixels from the bottom edge of the enclosing pane as when it was originally placed.
`sizFIXEDSTICKY`	The top and bottom edges stick to their original locations in the enclosing pane. If the enclosure scrolls vertically, the pane will scroll with it.
`sizELASTIC`	The height of the pane grows and shrinks by the same amount as the enclosing pane.

A couple of examples will make this clearer: A vertical scroll bar in a window would be horizontally `sizFIXEDRIGHT` and vertically `sizELASTIC`. It has a fixed horizontal length and remains anchored to the right edge of the window. Vertically, it stretches and contracts with the height of the window. A status box in the lower left corner of a window would be `sizFIXEDLEFT` horizontally and `sizFIXEDBOTTOM` vertically. It has a constant size and remains anchored to the bottom left corner of the window.

```
void IViewRes(ResType rType, short resID, CView *anEnclosure,
CBureaucrat *aSupervisor);
```

Initialize a pane from a resource template. `Rttype` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the same as for `IBorder()`. This method is inherited from CView.

To initialize a pane from a resource file, use a `'Pane'` resource.

```
void IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor, Ptr
viewData);
```

This method is used internally for initializing from a resource template. Each subclass of CView overrides this method to use its own resource template.

```
void Dispose(void);
```

Dispose of a pane. If you create a subclass of class CPane, be sure your class's `Dispose()` method calls `inherited::Dispose()`.

Accessing methods

```
void SetFrameOrigin(short fLeft, short fTop);
```

Set the coordinates of the top left corner of the pane's frame. The frame coordinates are the QuickDraw coordinates of the pane.

```
void GetFrame(Rect *theFrame);
```

Get the frame of the pane. The frame is the rectangle that encloses the pane in pane coordinates. Usually the top and left of this rectangle are 0,0.

```
void GetLengths(short *theWidth, short *theHeight);
```

Get the width and the height of the pane.

```
void GetOrigin(long *theHOrigin, long *theVOrigin);
```

Get the origin of a pane. The origin of a pane is the top left of the window in frame coordinates. It's the distance from the top left of the window to the (0,0) of the pane. You should not use or override this method.

```
void GetAperture(Rect *theAperture);
```

Get the aperture of the pane. The aperture is the visible portion of a pane. The aperture is the area where drawing can occur.

```
Boolean Contains(Point thePoint);
```

Returns TRUE if `thePoint` is in the pane. `ThePoint` is in window coordinates.

```
Boolean ReallyVisible(void);
```

Return TRUE is the pane is visible and within QuickDraw space. This method actually returns true if the pane and its enclosure is *potentially* visible. The pane may not actually be on the screen. Your pane is on the screen if it's `ReallyVisible()` and the aperture isn't empty.

```
void GetPixelExtent(long *hExtent, long *vExtent);
```

Get the dimensions of the pane in pixels. This is the same as the frame's width and height.

```
void SetPrintClip(ClipOption aPrintClip);
```

Specify the print option to use when printing. `APrintClip` can be one of

aPrintClip value	Meaning
`clipAPERTURE`	Print only what is visible
`clipFRAME`	Print the contents of the frame
`clipPAGE`	Print to fill the page

The default is `clipFRAME`.

Appearance methods

```
void Show(void);
```

Show the pane if it was hidden. The default method sends a `Refresh()` message to the pane after making it visible.

```
void Hide(void);
```

Hide the pane if it was visible. The default method sends a `Refresh()` message to the pane before hiding it.

Size and location methods

```
void Place(short hEncl, short vEncl, Boolean redraw);
```

Place the pane at the point `hEncl`, `vEncl` of its enclosure. `HEncl` and `vEncl` are given in the coordinate system of the enclosure. If `redraw` is TRUE, redraw the pane after moving it.

```
void Offset(short hOffset, short vOffset, Boolean redraw);
```

Offset the pane by `hOffset` pixels horizontally and `vOffset` pixels vertically. If `redraw` is TRUE, redraw the pane after moving it.

```
void ChangeSize(Rect *delta, Boolean redraw);
```

Change the size of the pane. The values of each field of the `delta` rectangle specify how each side should change. Positive values mean down and to the right. Negative values mean up and to the left.

```
void AdjustToEnclosure(Rect *deltaEncl);
```

Adjust the size or location of the pane when the enclosure has moved or changed size. You should not override or use this method.

```
void AdjustHoriz(Rect *deltaEncl, Rect *delta, short *offset,
Boolean *moved, Boolean *sized);
```

Adjust the horizontal size or location of a pane. You should not override or use this method.

```
void AdjustVert(Rect *deltaEncl, Rect *delta, short *offset,
Boolean *moved, Boolean *sized);
```

Adjust the vertical size or location of a pane. You should not override or use this method.

```
void EnclosureScrolled(short hOffset, short vOffset);
```

Adjust the location of a pane after its location has scrolled. This method affects the pane only if it is sticky in the direction it is being moved. You should not use or override this method.

Adapting methods

```
void FitToEnclosure(Boolean horizFit, Boolean vertFit);
```

Make the frame of the pane fit the interior of its enclosure in either the vertical or horizontal direction. If `horizFit` is TRUE, the left edge and width of the pane's frame change to coin-

cide with the enclosure's interior. If `vertFit` is TRUE, the top edge and height of the pane's frame change to coincide with the enclosure's interior. `FitToEnclosure()` sends the enclosure a `GetInterior()` message to determine the interior of the pane. `FitToEnclosure()` does not redraw the pane.

```
void FitToEnclFrame(Boolean horizFit, Boolean vertFit);
```

Fit the frame of the pane to the frame of its enclosure in either the vertical or horizontal direction. If `horizFit` is TRUE, the left edge and the width of the pane's frame change to coincide with the enclosure's frame. If `vertFit` is TRUE, the top edge and the height of the pane's frame change to coincide with the enclosure's frame. `FitToEnclFrame()` does not redraw the pane.

```
void CenterWithinEnclosure(Boolean horizCenter, Boolean
vertCenter);
```

Center the pane within its enclosure horizontally or vertically. Only the location of the pane changes. The size of the pane does not change. `CenterWithinEnclosure()` does not redraw the pane.

Drawing methods

```
void Draw(Rect *area);
```

Draw the contents of the pane. The `area` parameter specifies the portion of the pane that needs to be redrawn. `Area` is given in frame coordinates. The default method does nothing. Your subclass of pane must override this method if the pane actually displays data.

```
void DrawAll(Rect *area);
```

Draw the pane and all its subviews. Use this method when you want to force the entire pane to be redrawn without waiting for an update event. Scrolling is a good example. You want to redraw the pane as soon as it has scrolled instead of waiting for an update event. This method prepares the pane before sending `Draw()` messages to it and to its subpanes. You should not override this method. See `CPanorama::Scroll()` for an example of using `DrawAll()`.

```
void Refresh(void);
```

Force the pane to redraw itself on the next update event. Default method sends a `RefreshRect()` message to the pane using the frame as the area.

```
void RefreshRect(Rect *area);
```

Force a portion of the pane to redraw itself on the next update event. `Area` is given in frame coordinates. Only the portion of the pane that's actually visible will actually be redrawn.

Printing methods

```
void AboutToPrint(short *firstPage, short *lastPage);
```

Printing is about to begin. Your pane subclass can override this method to do anything that needs to happen right before printing.

```
void PrintPage(short pageNum, short pageWidth, short pageHeight);
```

Print the specified page of the pane. The default method ignores the page number and draws the entire pane. If your pane subclass supports multi-page documents, you must override this method to determine which part of the pane to draw.

```
void DonePrinting(void);
```

Printing is over. If you want to take any action when printing is done, override this method.

```
void PrepareToPrint(void);
```

Set up the coordinate system and the clipping region before printing. The default method sets the clipping region to the region described in the printClip variable. (You can set this variable with the `SetPrintClip()` method.)

Calibration methods

```
void Prepare(void);
```

Prepare the pane for drawing. `Prepare()` sets up the port and the QuickDraw coordinates for the pane. It also sets the clipping region to the aperture (the visible portion of the pane) so drawing is constrained to the visible portion of the pane. If the pane has an environment associated with it, `Prepare()` sends a `Restore()` message to the environment. If the pane is being printed, the default method sends the pane a `PrepareToPrint()` message instead.

```
void RestoreEnvironment(void);
```

Restore the pane's drawing environment. If the pane has a drawing environment associated with it, this method sends a `Restore()` message to it.

```
void CalcFrame(void);
```

Calculate the coordinates of the pane's frame based on the frame's width and height and its location within its enclosure. Generally, the superclasses of the pane classes you define will send this message. You should not use or override this method unless your pane subclass needs something other than (0,0) at its top left corner

```
void ResizeFrame(Rect *delta);
```

Adjust the frame when the size of the pane changes. The `delta` rectangle specifies how each side changes. Positive values mean down and to the right. Negative values mean up and to the left. The default method always sets the top left of the frame to (0,0)

Generally, you should not need to use or override this method unless your pane subclass requires that the top, left of the frame be something other than (0,0). See the implementation of `ResizeFrame()` in panorama for an example.

`void CalcAperture(void);`

Calculate the visible (or drawable) portion of a pane. The aperture of a pane is the area that is not obscured by the bounds of the enclosing view. You should not use or override this method. To get the aperture use `GetAperture()`.

Coordinate transformation methods

`void WindToFrame(Point *thePoint);`

Convert a point from window coordinates to frame coordinates.

`void WindToFrameR(Rect *theRect);`

Convert a rectangle from window coordinates to frame coordinates

`void FrameToWind(Point *thePoint);`

Convert a point from frame coordinates to window coordinates

`void FrameToWindR(Rect *theRect);`

Convert a rectangle from frame coordinates to window coordinates

`void EnclToFrame(Point *thePoint);`

Convert a point from the frame coordinates of its enclosure to the frame coordinates of the pane.

`void EnclToFrameR(Rect *theRect);`

Convert a rectangle from the frame coordinates of its enclosure to the frame coordinates of the pane.

`void FrameToEncl(Point *thePoint);`

Convert a point from frame coordinates to the frame coordinates of its enclosure.

`void FrameToEnclR(Rect *theRect);`

Convert a rectangle from frame coordinates to the frame coordinates of its enclosure.

`void FrameToGlobalR(Rect *theRect);`

Convert a rectangle from the pane's frame coordinates to global coordinates.

CPane summary

```
typedef enum SizingOption      {
    sizFIXEDLEFT,    /* Fixed length, anchored to left          */
    sizFIXEDRIGHT,        /* Fixed length, anchored to right           */
    sizFIXEDTOP,     /* Fixed length, anchored to top     */
    sizFIXEDBOTTOM,       /* Fixed length, anchored to bottom          */
    sizFIXEDSTICKY,       /* Fixed length, sticks to coords of its enclosure */
    sizELASTIC       /* Variable length, always a fixed amount smaller than enclosure */
} SizingOption;

typedef enum ClipOption        {
    clipAPERTURE,
    clipFRAME,
    clipPAGE
} ClipOption;


struct CPane : CView {
    short           width;
    short           height;
    short           hEncl;
    short           vEncl;
    SizingOption    hSizing;
    SizingOption    vSizing;
    Boolean         autoRefresh;
    Rect            frame;
    Rect            aperture;
    long            hOrigin;
    long            vOrigin;
    CEnvironment    *itsEnvironment;
    ClipOption      printClip;
    Boolean         printing;


    void      IPane(CView *anEnclosure, CBureaucrat *aSupervisor,
                    short aWidth, short aHeight, short aHEncl, short aVEncl,
                    SizingOption aHSizing, SizingOption aVSizing);
    void      IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor,
                        Ptr viewData);
    void      IPaneX(void);
    void      Dispose(void);

    void      SetFrameOrigin(short fLeft, short fTop);
    void      GetFrame(Rect *theFrame);
    void      GetLengths(short *theWidth, short *theHeight);
    void      GetOrigin(long *theHOrigin, long *theVOrigin);
    void      GetAperture(Rect *theAperture);
    Boolean   Contains(Point thePoint);
    Boolean   ReallyVisible(void);
    void      GetPixelExtent(long *hExtent, long *vExtent);
    void      SetPrintClip(ClipOption aPrintClip);

    void      Show(void);
    void      Hide(void);

    void      Place(short hEncl, short vEncl, Boolean redraw);
    void      Offset(short hOffset, short vOffset, Boolean redraw);
    void      ChangeSize(Rect *delta, Boolean redraw);
    void      AdjustToEnclosure(Rect *deltaEncl);
    void      AdjustHoriz(Rect *deltaEncl, Rect *delta, short *offset,
                          Boolean *moved, Boolean *sized);
    void      AdjustVert(Rect *deltaEncl, Rect *delta, short *offset,
                         Boolean *moved, Boolean *sized);
```

```
    void      EnclosureScrolled(short hOffset, short vOffset);

    void      FitToEnclosure(Boolean horizFit, Boolean vertFit);
    void      FitToEnclFrame(Boolean horizFit, Boolean vertFit);
    void      CenterWithinEnclosure(Boolean horizCenter, Boolean vertCenter);

    void      Draw(Rect *area);
    void      DrawAll(Rect *area);
    void      Refresh(void);
    void      RefreshRect(Rect *area);

    void      AboutToPrint(short *firstPage, short *lastPage);
    void      PrintPage(short pageNum, short pageWidth, short pageHeight);
    void      DonePrinting(void);
    void      PrepareToPrint(void);

    void      Prepare(void);
    void      RestoreEnvironment(void);
    void      CalcFrame(void);
    void      ResizeFrame(Rect *delta);
    void      CalcAperture(void);

    void      WindToFrame(Point *thePoint);
    void      WindToFrameR(Rect *theRect);
    void      FrameToWind(Point *thePoint);
    void      FrameToWindR(Rect *theRect);
    void      EnclToFrame(Point *thePoint);
    void      EnclToFrameR(Rect *theRect);
    void      FrameToEncl(Point *thePoint);
    void      FrameToEnclR(Rect *theRect);
    void      FrameToGlobalR(Rect *theRect);
};
```

CPanorama
41

Introduction

CPanorama is an abstract class for implementing displays that may be larger than the frame of a pane. The frame is a viewport though which a portion of the panorama is visible.

Heritage

Superclass	CPane
Subclasses	CPicture
	CStaticText

Using CPanorama

Use a subclass of CPanorama whenever you want to display something that is larger than the pane you want to view it through. For example, some pictures are much bigger than a standard Macintosh screen. For an example of a panorama, look at the CStaticText class. In almost every case, you'll use a panorama with a scroll pane (see CScrollPane for a description).

Drawing in a panorama is almost the same as drawing in a pane. The main difference is that panoramas can have their own coordinate system. Each panorama unit can map to one or more pixels. For example, in the CStaticText class, the horizontal unit is set to the number of pixels in the widest character, and the vertical unit is set to the number of pixels of the height of a line. Scroll panes use this information to set the scroll bars accurately.

> **Note:** Panoramas do not support fractional scales or non-linear scales for coordinate systems.

The size of the panorama image is called the **bounds**. In most cases, the top, left corner of the bounds is the point (0, 0), but it's not required. The way to think of the relationship between the panorama and the frame of the pane is that the panorama image is stationary as the frame roves over it. As you move around in a panorama, the coordinates of the top, left corner of the frame will change.

Variables

`Rect bounds;`	Bounds defining Pane coordinates
`Point position;`	Location of frame in panorama
`short hScale;`	Pixels per horizontal unit
`short vScale;`	Pixels per vertical unit
`Point savePosition;`	Save for later restoration
`CScrollPane *itsScrollPane;`	Scroll pane a panorama belongs to, if any

Methods

Construction and destruction methods

```
void IPanorama(CView *anEnclosure, CBureaucrat *aSupervisor, short
aWidth, short aHeight, short aHEncl, short aVEncl, SizingOption
aHSizing, SizingOption aVSizing);
```

Initialize a panorama. The arguments to this routine are identical to the ones for `IPane()`.

> **Note:** The descriptions of the other arguments are in CPane.

```
void IViewRes(ResType rType, short resID, CView *anEnclosure,
CBureaucrat *aSupervisor);
```

Initialize a panorama from a resource template. `RTtype` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the same as for `IBorder()`. This method is inherited from CView.

To initialize a panorama from a resource file, use a `'Pano'` resource.

```
void IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor, Ptr
viewData);
```

This method is used internally for initializing from a resource template. Each subclass of CView overrides this method to use its own resource template.

Accessing methods

```
void GetExtent(long *theHExtent, long *theVExtent);
```

Get the size of each side of the panorama in panorama units.

```
void GetFramePosition(long *theHPos, long *theVPos);
```

Determine how far the top, left of the frame is from the top, left of the bounds in panorama coordinates. The CScrollPane class uses this method to figure out the maximum settings for the scroll bars. You should not use or override this method.

```
void GetFrameSpan(short *theHSpan, short *theVSpan);
```
Return the number of panorama units that the frame spans.

```
void SetBounds(Rect *aBounds);
```
Set the bounds of the panorama. The bounds define the size of the data displayed in the panorama and the panorama coordinates. If the panorama's enclosure is a scroll pane, this method sends it an `AdjustScrollMax()` message to adjust the scroll bars.

```
void GetBounds(Rect *theBounds);
```
Get the bounds of a panorama.

```
void SetPosition(Point aPosition);
```
Set the position of the frame in relation to the panorama. The point is in panorama coordinates. If the panorama's enclosure is a scroll pane, this method sends it a `Calibrate()` message to adjust the position of scroll box (the scroll bars' "thumb").

```
void GetPosition(Point *thePosition);
```
Get the position of the frame in relation to the panorama. In other words, what are the panorama coordinates of the top, left of the frame?

```
void SetScales(short aHScale, short aVScale);
```
Set the horizontal and vertical scales. Specify the scales in pixels per panorama unit. For instance, the CStaticText class (a subclass of CPanorama) uses the width of the widest character as its horizontal unit and the height of a line as a vertical unit. If the panorama's enclosure is a scroll pane, this method sends it an `AdjustScrollMax()` message to adjust the scroll bars.

```
void GetScales(short *theHScale, short *theVScale);
```
Get the scale factors of the panorama.

```
void SetScrollPane(struct CScrollPane *aScrollPane);
```
Specify the scroll pane that controls this panorama.

```
void GetHomePosition(Point *theHomePos);
```
Get the location of the top, left corner of the panorama in panorama coordinates.

```
void GetPixelExtent(long *hExtent, long *vExtent);
```
Get the size of each side of the panorama in pixels.

Calibrating methods

```
void ResizeFrame(Rect *delta);
```

Adjust the frame of a panorama when its size changes. The `delta` rectangle specifies the amount of change for each side. Positive numbers mean down and to the right. Negative numbers mean up and to the left.

Scrolling methods

```
void Scroll(short hDelta, short vDelta, Boolean redraw);
```

Scroll a panorama by `hDelta` units horizontally and `vDelta` units vertically. The units are given in panorama coordinates.

```
void ScrollTo(Point aPosition, Boolean redraw);
```

Scroll the panorama to a specific position. The units given in `aPosition` are in panorama coordinates.

```
void ScrollToSelection(void);
```

Scroll the panorama so the selection is visible. The default method does nothing. Your panorama subclass must override this method.

```
Boolean AutoScroll(Point mouseLoc);
```

Scroll automatically during a mouse-down. Returns TRUE if scrolling actually took place. Typically, you would send this message in the same routine that tracks selection. See the implementation of this method in the CEditText class for an example.

Printing methods

```
void AboutToPrint(short *firstPage, short *lastPage);
```

The specified range of pages are about to be printed. You can override this method if your subclass needs to take some action before printing.

```
void PrintPage(short pageNum, short pageWidth, short pageHeight);
```

Print the specified page. `PageWidth` is the width of the page in pixels. `PageHeight` is the height of the page in pixels.

```
void DonePrinting(void);
```

Printing has stopped. Override this method if you need to do some cleanup after printing.

CPanorama summary

```
struct CPanorama : CPane {

    Rect            bounds;
    short           hScale;
    short           vScale;
    Point           position;
    Point           savePosition;
    struct CScrollPane    *itsScrollPane;



    void            IPanorama(CView *anEnclosure, CBureaucrat *aSupervisor,
                              short aWidth, short aHeight,
                              short aHEncl, short aVEncl,
                              SizingOption aHSizing, SizingOption aVSizing);
    void            IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor,
                              Ptr viewData);


    void            GetExtent(long *theHExtent, long *theVExtent);
    void            GetFramePosition(long *theHPos, long *theVPos);
    void            GetFrameSpan(short *theHSpan, short *theVSpan);
    void            SetBounds(Rect *aBounds);
    void            GetBounds(Rect *theBounds);
    void            SetPosition(Point aPosition);
    void            GetPosition(Point *thePosition);
    void            SetScales(short aHScale, short aVScale);
    void            GetScales(short *theHScale, short *theVScale);
    void            SetScrollPane(struct CScrollPane *aScrollPane);

    void            GetHomePosition(Point *theHomePos);
    void            GetPixelExtent(long *hExtent, long *vExtent);


    void            ResizeFrame(Rect *delta);


    void            Scroll(short hDelta, short vDelta, Boolean redraw);
    void            ScrollTo(Point aPosition, Boolean redraw);
    void            ScrollToSelection(void);
    Boolean         AutoScroll(Point mouseLoc);


    void            AboutToPrint(short *firstPage, short *lastPage);
    void            PrintPage(short pageNum, short pageWidth, short pageHeight);
    void            DonePrinting(void);
};
```

CPicture
42

Introduction

CPicture lets you display Macintosh pictures. The pictures are not editable.

Heritage

Superclass	CPanorama
Subclasses	None

Using CPicture

Use CPicture when you want to display a Macintosh picture. You can display the picture in a scroll frame or you can scale it to fit its frame. The picture is not editable. The picture is a standard Macintosh picture and is stored in an instance variable. You can use this class to learn how to create your own panorama subclasses.

Variables

`PicHandle macPicture`	Handle to the QuickDraw picture
`Boolean scaled`	TRUE if picture is scaled to fit in frame

Methods

Construction/Destruction

```
void IPicture(CView *anEnclosure, CBureaucrat *aSupervisor, short
aWidth, short aHeight, short aHEncl, short aVEncl, SizingOption
aHSizing, SizingOption aVSizing);
```

Initialize a picture. The arguments are identical to pane initialization.

Note: The descriptions of the other arguments are in CPane.

```
void IViewRes(ResType rType, short resID, CView *anEnclosure,
CBureaucrat *aSupervisor);
```

Initialize a picture from a resource template. `RTtype` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure`

and aSupervisor are the same as for IPicture(). This method is inherited from CView.

To initialize a picture from a resource file, use a 'PctP' resource.

Appearance methods

```
void Draw(Rect *area);
```

Draw the picture. This method ignores the area parameter.

Accessing methods

```
void SetMacPicture(PicHandle aMacPicture);
```

Use aMacPicture as the Macintosh picture for this object. This method makes the aMacPicture handle unpurgeable. To release the memory that the picture occupies, use GetMacPicture() to get the PicHandle, and call one of the Toolbox routines ReleaseResource()

```
void UsePICT(short PICTid);
```

Use the PICT resource with ID PICTid as the Macintosh picture for this object.

```
PicHandle GetMacPicture(void);
```

Return a handle to the Macintosh picture.

Calibration methods

```
void SetScaled(Boolean aScaled);
```

If aScaled is TRUE, the picture will be scaled to fit its frame.

```
Boolean GetScaled(void);
```

Return TRUE if the picture is scaled.

```
void ResizeFrame(Rect *delta);
```

Resize the picture's frame by the amount specified.

```
void FrameToBounds(void);
```

Make the frame of the picture the same size as the bounds.

```
void GetExtent(long *theHExtent, long *theVExtent);
```

Return the size of the picture. The size will be different if the picture is scaled.

CPicture summary

```
struct CPicture : CPanorama {
    PicHandle       macPicture;
    Boolean         scaled;

    void            IPicture(CView *anEnclosure, CBureaucrat *aSupervisor,
                             short aWidth, short aHeight,
                             short aHEncl, short aVEncl,
                             SizingOption aHSizing, SizingOption aVSizing);

    void            Draw(Rect *area);

    void            SetMacPicture(PicHandle aMacPicture);
    void            UsePICT(short PICTid);
    PicHandle       GetMacPicture(void);
    void            SetScaled(Boolean aScaled);
    Boolean         GetScaled(void);
    void            ResizeFrame(Rect *delta);
    void            FrameToBounds(void);
    void            GetExtent(long *theHExtent, long *theVExtent);
};
```

CPrinter
43

Introduction

CPrinter is a class that handles standard Macintosh printing dialogs and calls the appropriate Print Manager routines.

Heritage

Superclass	CObject
Subclasses	None

Using CPrinter

The printer object manages communication between a document and the Macintosh print manager. Every document object can have a printer object associated with it. The document's `PrintPageOfDoc()` method is the method that actually does the printing.

If you like, you can store a Macintosh print record with your document to preserve defaults. You can pass a handle to this record to the `IPrinter()` method when you initialize your document. For methods in this class that return a Boolean value, TRUE means that the printer record stored with the printer object has changed.

Ordinarily, you won't need to create a subclass of this class.

Variables

`CDocument *itsDocument;`	Document using this Printer
`THPrint macTPrint;`	Toolbox print record

Methods

Construction and destruction methods

```
Boolean IPrinter(CDocument *aDocument, THPrint aMacTPrint);
```

Initialize a printer object. `ADocument` is the document the printer object is associated with. `AMacTPrint` is a Macintosh `THPrint` record handle. If `aMacTPrint` is NULL, this method creates a new `THPrint` record. This method returns TRUE if it had to update the `aMacTPrint` record because it was incompatible. The document's initialization method initializes a printer object automatically if the printable parameter to `IDocument()` is TRUE.

```
void Dispose(void);
```

Dispose of a printer object.

Accessing methods

```
Boolean OpenPrintMgr(void);
```

Open the Print Manager to get ready to print. This method return TRUE if there was no error. If there was an error, this method displays an alert that tells the user to choose a printer with the Chooser, first.

```
THPrint GetPrintRecord(void);
```

Return the Toolbox print record. This method calls `PrValidate()` to update the record. You should treat the value that this method returns as read-only.

```
void GetPageInfo(Rect *paperRect, Rect *pageRect, short *hRes,
short *vRes);
```

Get information about the paper size and printable area of the page. The `paperRect` and `pageRect` are specified in dots. `HRes` and `vRes` specify the number of dots per inch.

```
Boolean DoPageSetup(void);
```

Respond to a **Page Setup** menu command. This method displays the standard Page Setup dialog, and returns TRUE if you made changes and pressed the OK button.

Printing methods

```
void DoPrint(void);
```

Respond to a **Print** menu command. This method displays the standard print job dialog. If the user clicks on the OK button, this method sends a `PrintPageRange()` message.

```
void PrintPageRange(short firstPage, short lastPage);
```

Print the specified range of the document associated with this printer object. The `DoPrint()` method usually sends this message. Your application can bypass the print dialogs and send this message directly, but TechNote 122 discourages this practice.

This method sends your document an `AboutToPrint()` message to give it an opportunity to adjust the page range. Then, for each page in the page range, it sends your document a `PrintPageofDoc()` message.

CPrinter summary

```
struct CPrinter : CObject {
    CDocument       *itsDocument;
    THPrint         macTPrint;

    Boolean         IPrinter(CDocument *aDocument, THPrint aMacTPrint);
    void            Dispose(void);

    Boolean         OpenPrintMgr(void);
    THPrint         GetPrintRecord(void);
    void            GetPageInfo(Rect *paperRect, Rect *pageRect,
                                    short *hRes, short *vRes);
    Boolean         DoPageSetup(void);

    void            DoPrint(void);
    void            PrintPageRange(short firstPage, short lastPage);
};
```

CRadioButton
44

Introduction

CRadioButton implements a standard Macintosh radio button.

Heritage

Superclass	CButton
Subclasses	None

Using CRadioButton

CRadioButton is a class that implements the standard Macintosh radio button. Since radio buttons always come in groups, a button must be a part of a radio group. The class CRadioGroup implements radio groups.

Like any other button, radio buttons can have a command associated with it. Use the `SetClickCmd()` method to set a radio button's command. You can also use any of the other CControl methods to manipulate the radio button.

Variables

`short idNumber;`	ID number for the button, usually set to the `CNTL` resource ID of the button. CRadioGroup uses the id number to keep track of which button is highlighted.

Methods

```
void IRadioButton(short CNTLid, CView *anEnclosure, CBureaucrat
*aSupervisor);
```

Initialize a radio button. `CNTLid` is the resource ID for the radio button. `AnEnclosure` is the window or pane the radio appears in. `ASupervisor` is the supervisor of the radio button. It must be a member of class CRadioGroup. The radio group uses the `CNTLid` as the button's id.

```
void DoGoodClick(short whichPart);
```

When the user presses and releases the mouse within the radio button, and the radio button was off, this method toggles the value of the radio button. This method sends a `ChangeStation()` message to the radio button's supervisor (a radio group).

If the control has a command number associated with it, this method sends a `DoCommand()` message to its supervisor.

```
short GetIdNumber(void);
```

Returns the ID number of the radio button.

CRadioButton summary

```
struct CRadioButton : CButton {
   short    idNumber;

   void     IRadioButton(short CNTLid, CView *anEnclosure, CBureaucrat *aSupervisor);
   void     DoGoodClick(short whichPart);
   short    GetIdNumber(void);
};
```

CRadioGroup
45

Introduction

CRadioGroup is a class that manages a group of radio buttons.

Heritage

Superclass	CBureaucrat
Subclasses	None

Using CRadioGroup

Use a radio group when you want to work with a group of radio buttons. According to the Macintosh human interface guidelines, only one button in a radio group can be on at a time. This class handles radio buttons so one turns off when you turn another one on.

The button that is on in a radio group is called the **station**. Every button in a radio group has an ID number. By default, this number is the same as the CNTL ID that was used to create it. You can set and get the station by object reference or by ID.

Variables

`CCluster *itsButtons`	Radio buttons belonging to this group
`CRadioButton *station`	The currently selected radio button

Methods

Initialization methods

```
void IRadioGroup(CBureaucrat *aSupervisor);
```

Initialize a radio group. `ASupervisor` is the bureaucrat that owns the radio group. Typically, the supervisor is a pane or a window.

Insertion and deletion methods

```
void AddButton(CRadioButton *theRadioButton);
```

Add a radio button to the group.

`void RemoveButton(CRadioButton *theRadioButton);`
Remove a radio button from the group.

`void AddButtonID(short CNTLid, CView *anEnclosure);`
Add a radio button, specified by a `CNTL` resource ID, to the group The radio group will be the radio button's supervisor. `AnEnclosure` is the pane or window it belongs to.

Accessing methods

`void ChangeStation(CRadioButton *theStation);`
Change the current selection to the specified button. Turn off the old selection and turn on this one.

`CRadioButton* GetStation(void);`
Return a reference to the currently selected radio button object. Returns NULL if no station is selected

`void SetStationID(short aStationID);`
Change the current selection to the button with the specified ID number.

`short GetStationID(void);`
Return the ID number of the currently selected radio button. Returns `NO_STATION` if no station is selected.

CRadioGroup summary

```
struct CRadioGroup : CBureaucrat {
    CCluster        *itsButtons;
    CRadioButton    *station;

    void                IRadioGroup(CBureaucrat *aSupervisor);
    void                Dispose(void);

    void                AddButton(CRadioButton *theRadioButton);
    void                RemoveButton(CRadioButton *theRadioButton);
    void                AddButtonID(short CNTLid, CView *anEnclosure);
    void                ChangeStation(CRadioButton *theStation);
    CRadioButton*   GetStation(void);
    void                SetStationID(short aStationID);
    short               GetStationID(void);
};
```

This is the value that `GetStationID()` returns when no station is selected.

```
#define     NO_STATION          0     /* Indicates no station is selected          */
```

CScrollBar
46

Introduction

CScrollBar implements a standard Macintosh scroll bar.

Heritage

Superclass	CControl
Subclasses	None

Using CScrollBar

This class implements a standard Macintosh scroll bar. To make scroll bars easier to use, this class distinguishes between a mouse click in an indicator (the scroll box or the "thumb") and a click in any other part of the scroll bar. Both behaviors are implemented in the `DoClick()` method of the CControl class.

When you click in any part other than an indicator, the `DoClick()` method calls the Toolbox routine `TrackControl()` with an action procedure, or **action proc**. The action procedure is the routine that adjusts what the scroll bar controls. In most cases, the scroll bar controls a panorama. To set the action proc use the `SetActionProc()` method inherited from CControl.

When you click in the indicator, the `DoClick()` method sends a `DoThumbDragged()` message to the scroll bar. This method calls a **thumb function** that you provide. The thumb function is the routine that adjusts whatever the scroll bar controls. Thumb functions are unique to scroll bars. To set the thumb function, use the `SetThumbFunc()` method.

Usually, you'll use a scroll bar to control a pane. The class CScrollPane is a scrollable pane (a panorama) with one or two scroll bars. The CScrollPane class handles the usual cases, so you don't have to provide an action proc or a thumb function.

Variables

`Orientation theOrientation`	Horizontal or Vertical
`VoidFunc theThumbFunc`	Function to call after a thumb drag

Methods

Construction and destruction methods

```
void IScrollBar(CView *anEnclosure, CBureaucrat *aSupervisor,
Orientation anOrientation, short aLength, short aHEncl, short
aVEncl);
```

Initialize a scroll bar. `AnEnclosure` is the pane or window the scroll bar belongs to. `ASupervisor` is the scroll bar's supervisor in the chain of command. `Orientation` is either `HORIZONTAL` or `VERTICAL`. `ALength` is the length of the scroll bar. `AHEncl` and `aVEncl` are the horizontal and vertical position of the upper left corner of the scroll bar.

Accessing methods

```
void SetThumbFunc(VoidFunc aThumbFunc);
```

Set `aThumbFunc` to be the scroll bar's thumb function. The default `DoClick()` method for controls sends a `DoThumbDragged()` message to the control when the user moves an indicator in a control. The `DoThumbDragged()` method for scroll bars calls the thumb function. You should declare the thumb function like this:

```
MyThumbFunc (CControl *theControl, short delta)
```

`TheControl` is the control whose indicator was moved. `Delta` is the amount by which the value changed. To get the current value of the control, you can send it a `GetValue()` message.

Drawing methods

```
void Draw(Rect *area);
```

Draw the scroll bar. If the scroll bar is active, this method draws it the normal way. If the scroll bar is inactive, this method draws only the frame of the scroll bar.

```
void Activate(void);
```

Activate the scroll bar.

```
void Deactivate(void);
```

Deactivate the scroll bar.

Click Response methods

```
void DoClick(Point hitPt, short modifierKeys, long when);
```

Handle a click in the scroll bar. If the scroll bar belongs to a scroll pane, send an `AdjustScrollMax()` message to the scroll pane.

```
void DoThumbDragged(short delta);
```

If the scroll bar has a thumb function associated with it, call it with the scroll bar and delta as arguments. The default `DoClick()` method for controls sends a `DoThumbDragged()`

message to the control when the user moves an indicator in a control. See SetThumbFunc() above.

CScrollBar summary

```
#define       SBARSIZE          16    /* Size of a scroll bar in pixels          */
#define       SBARSIZE1         15    /* SBARSIZE - 1                                    */

typedef enum {
    HORIZONTAL,
    VERTICAL
} Orientation;

struct CScrollBar : CControl {
    Orientation     theOrientation;
    VoidFunc        theThumbFunc;

    void      IScrollBar(CView *anEnclosure, CBureaucrat *aSupervisor,
                         Orientation anOrientation, short aLength,
                         short aHEncl, short aVEncl);

    void      SetThumbFunc(VoidFunc aThumbFunc);

    void      Draw(Rect *area);
    void      Activate(void);
    void      Deactivate(void);

    void      DoClick(Point hitPt, short modifierKeys, long when);
    void      DoThumbDragged(short delta);

};
```

CScrollPane
47

Introduction

CScrollPane implements a pane with scroll bars to control a panorama.

Heritage

Superclass	CPane
Subclasses	None

Using CScrollPane

A scroll pane is a pane with a panorama and scroll bars to control what is being displayed in the pane. Most of your applications will use a scroll pane that occupies most of the window. After creating a scroll pane, you might want to send it a `FitToEnclFrame()` message to make it as big as the window.

All you have to do to use a scroll pane is install a panorama with the `InstallPanorama()` method. The scroll pane uses the scale of the panorama for the values of the scroll bars.

The scroll bars and panorama do not communicate directly. Mouse clicks in the scroll bars are reported to the scroll pane, which then tells the panorama how to scroll or shift its image. Similarly, changes in the panorama which would affect the scroll bars are reported to the scroll pane, which then adjust the scroll bars.

Variables

`CPanorama *itsPanorama;`	The scrollable view. The "content" of the scroll pane
`CScrollBar *itsHorizSBar;`	The scroll pane's horizontal scroll bar
`CScrollBar *itsVertSBar;`	The scroll pane's vertical scroll bar
`CSizeBox *itsSizeBox`	The scroll pane's size box
`long hExtent;`	For internal use
`long vExtent;`	For internal use
`short hUnit;`	For internal use
`short vUnit;`	For internal use
`short hSpan;`	For internal use
`short vSpan;`	For internal use

`short hStep;`	Number of horizontal units to scroll by when the user clicks on an arrow.
`short vStep;`	Number of vertical units to scroll by when the user clicks on an arrow.
`short hOverlap;`	Number of units to overlap when the user clicks in a page region.
`short vOverlap;`	Number of units to overlap when the user clicks in a page region.

Methods

Construction and destruction methods

```
void IScrollPane(CView *anEnclosure, CBureaucrat *aSupervisor,
short aWidth, short aHeight, short aHEncl, shore aVEncl,
SizingOption aHSizing, SizingOption aVSizing, Boolean hasHoriz,
Boolean hasVert, Boolean hasSizeBox)
```

Initialize a scroll pane. All but the last three arguments are identical to the arguments to `IPane()`. If `hasHoriz` is TRUE, the scroll pane has an horizontal scroll bar. If `hasVert` is TRUE, the scroll pane has a vertical scroll bar. If `hasSizeBox` is TRUE, the scroll pane will draw a size box in the lower right corner of the pane.

> **Note:** The descriptions of the other arguments are in CPane.

```
void IViewRes(ResType rType, short resID, CView *anEnclosure,
CBureaucrat *aSupervisor);
```

Initialize a scroll pane from a resource template. `RTtype` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the same as for `IScrollPane()`. This method is inherited from CView.

To initialize a scroll pane from a resource file, use a `'ScPn'` resource.

```
void IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor, Ptr
viewData);
```

This method is used internally for initializing from a resource template. Each subclass of CView overrides this method to use its own resource template.

Accessing methods

```
void InstallPanorama(CPanorama *aPanorama);
```

Establish `aPanorama` as the panorama associated with this scroll pane. This *method* sends `AdjustScrollMax()` and `Calibrate()` messages to the scroll pane.

```
void SetSteps(short aHStep, short aVStep);
```
Set the amount to scroll when the user clicks on the arrows of a scroll bar. The units are in the panorama's units.

```
void GetSteps(short *aHStep, short *aVStep);
```
Get the amount to scroll when the user clicks on the arrows of a scroll bar. The units are in the panorama's units.

```
void SetOverlaps(short aHOverlap, short aVOverlap);
```
Set the amount of overlap when the user clicks in the page (gray) regions of the scroll bar. The units are in the panorama's units.

```
void GetInterior(Rect *theInterior);
```
Get the interior of the scroll pane. The interior excludes the space the scroll bars occupy.

Scroll bar maintenance methods

```
void AdjustScrollMax(void);
```
Adjust the maximum value of the scroll bars from the extent and frame size of the panorama.

```
void Calibrate(void);
```
Adjust the scroll bar's thumb when the position of the frame of the panorama changes.

```
void ChangeSize(Rect *delta, Boolean redraw);
```
Change the size of a scroll pane. Each component of the `delta` rectangle specifies how each side will change. Positive values mean down and to the right. Negative values mean up and to the left. If redraw is TRUE, the scroll pane is redrawn on the next update event.

Scroll performance methods

```
void DoHorizScroll(short whichPart);
```
Scroll horizontally. `WhichPart` specifies which part of the scroll bar was hit. This method sends a `Scroll()` message to the panorama.

```
void DoVertScroll(short whichPart);
```
Scroll vertically. `WhichPart` specifies which part of the scroll bar was hit. This method sends a `Scroll()` message to the panorama.

```
void DoThumbDrag(short hDelta, short vDelta);
```
Adjust the panorama when the scroll box (thumb) has been dragged. This method sends a `Scroll()` message to the panorama.

Scroll action functions

Note that these are functions, not methods.

```
pascal void SBarActionProc(ControlHandle macControl, short
whichPart);
```

The Toolbox `TrackControl()` routine calls this function continuously while the mouse is down in any part of the scroll bar except the scroll box (thumb). This function sends the scroll bar's supervisor (the scroll pane) `DoHorizScroll()` and `DoVertScroll()` messages that actually scroll the panorama.

```
void SBarThumbFunc (CScrollBar *theSBar, short delta);
```

The scroll bar's `DoThumbDragged()` method calls this routine after the scroll box of a scroll bar has been moved. A control's `DoClick()` method sends a `DoThumbDragged()` message when the user moves the indicator of a control. This function sends the scroll bar's supervisor (the scroll pane) a `DoThumbDrag()` message.

CScrollPane summary

```
struct CScrollPane : CPane {
    CPanorama       *itsPanorama;
    CScrollBar      *itsHorizSBar;
    CScrollBar      *itsVertSBar;
    CSizeBox  *itsSizeBox;
    long      hExtent;
    long      vExtent;
    short     hUnit;
    short     vUnit;
    short     hSpan;
    short     vSpan;
    short     hStep;
    short     vStep;
    short     hOverlap;
    short     vOverlap;

    void      IScrollPane(CView *anEnclosure, CBureaucrat *aSupervisor, short aWidth,
                          short aHeight, short aHEncl, short aVEncl,
                          SizingOption aHSizing, SizingOption aVSizing,
                          Boolean hasHoriz, Boolean hasVert, Boolean hasSizeBox);
    void      IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor,
    Ptr viewData);
    void      IScrollPaneX(Boolean hasHoriz, Boolean hasVert, Boolean hasSizeBox);

    void      InstallPanorama(CPanorama *aPanorama);
    void      SetSteps(short aHStep, short aVStep);
    void      GetSteps(short *theHStep, short *theVStep);
    void      SetOverlaps(short aHOverlap, short aVOverlap);
    void      GetInterior(Rect *theInterior);

    void      AdjustScrollMax(void);
    void      Calibrate(void);
    void      ChangeSize(Rect *delta, Boolean redraw);

    void      DoHorizScroll(short whichPart);
    void      DoVertScroll(short whichPart);
    void      DoThumbDrag(short hDelta, short vDelta);
};
```

CSizeBox
48

Introduction

CSizeBox implements a Macintosh grow icon.

Heritage

Superclass	CPane
Subclasses	None

Using CSizeBox

In most cases, you will not need to use this class yourself. It is used by CScrollPane to draw a Macintosh grow icon at the lower left corner of any pane, not just in the lower right corner of a window.

Variables

This class has no instance variables

Methods

Construction and destruction methods

```
void ISizeBox(CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a size box. This method places a pane containing a size box at the lower right of its enclosure. The sizing options for a size box are `sizFIXEDRIGHT` and `sizFIXEDBOTTOM`.

```
void Dispose();
```

Dispose of a size box.

Appearance methods

```
void Draw(Rect *area);
```

Draw a size box. If the pane is active, this method draws the size box. If the pane is inactive, this method draws a white rectangle.

`void Activate(void);`

The enclosure the size box belongs to is becoming active. The default method sends a `Draw()` message to the size box

`void Deactivate(void);`

The enclosure the size box belongs to is becoming inactive. The default method sends a `Draw()` message to the size box.

Class resources

This class uses an `SICN` resource to draw the grow icon so it can be drawn anywhere. The standard Macintosh grow icon always appears in the lower right corner of a window.

`SICN 200`	The grow icon

CSizeBox summary

```
struct CSizeBox : CPane {

    void      ISizeBox(CView *itsEnclosure, CBureaucrat *itsSupervisor);
    void      Dispose(void);

    void      Draw(Rect *area);
    void      Activate(void);
    void      Deactivate(void);
};
```

CStaticText 49

Introduction

CStaticText implements a pane that displays non-editable text. This class uses the Macintosh Text Edit routines.

Heritage

Superclass	CPanorama
Subclasses	CEditText

Using CSaticText

You can use objects of this class any time you need to display non-editable text. static text pane is usually the panorama in a scroll pane so you can scroll through the text. A good application for this class would be a help window.

Because CStaticText is based on the Macintosh Text Edit (TE) routines, it has some limitations. The TE routines were designed to display small amounts of text. The maximum number of characters you can store in a CStaticText record is around 32,000, but you'll notice performance degradation long before it gets that big.

> **Note:** CStaticText does not use the Styled Text Edit routines described in *Inside Macintosh V.*

Variables

`TEHandle macTE`	Toolbox Text Edit record handle
`short lineWidth`	Width of a text line in pixels. If <= 0, width is the same as that of the TE viewRect
`Boolean wholeLines;`	If TRUE, draw only whole lines, don't cut off lines vertically
`long spacingCmd`	Line spacing command number

Methods

Construction and destruction methods

```
void IStaticText(CView *anEnclosure, CBureaucrat *aSupervisor,
short aWidth, short aHeight, short aHEncl, short aVEncl,
SizingOption aHSizing, SizingOption aVSizing, short aLineWidth);
```

Initialize an static text pane. Most of the arguments to this method are identical to the pane initialization. `ALineWidth` specifies how wide the lines should be. If it's less than zero, the width is the same as the Macintosh TE record's `viewRect`.

> **Note:** The descriptions of the other arguments are in CPane.

```
void IViewRes(ResType rType, short resID, CView *anEnclosure,
CBureaucrat *aSupervisor);
```

Initialize a static text pane from a resource template. `RTtype` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the same as for `IStaticText()`. This method is inherited from CView.

To initialize a static text pane from a resource file, use a `'StTx'` resource.

```
void IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor, Ptr
viewData);
```

This method is used internally for initializing from a resource template. Each subclass of CView overrides this method to use its own resource template.

```
void Dispose(void);
```

Dispose of a static text object.

Display methods

```
void Draw(Rect *area);
```

Draw the static text object.

```
void Scroll(short hDelta, short vDelta, Boolean redraw);
```

Scroll the static text object by `hDelta` units (characters) horizontally and `vDelta` units (lines) vertically. If `redraw` is TRUE, force a redisplay at the next update event.

```
void ScrollToSelection(void);
```

Scroll so the current selection is visible.

Printing methods

```
void PrintPage(short pageNum, short pageWidth, short pageHeight);
```

Print the specified page. This method is usually called by the document's `PrintPageOfDoc()` method. It supplies the values for `pageWidth` and `pageHeight`.

```
void DonePrinting(void);
```

Printing has been completed.

Text specification methods

```
void SetTextString(Str255 textStr);
```

Use the specified string as the static text object's text.

```
void SetTextHandle(Handle textHand);
```

Use the text `textHand` as the text for this static text object.

```
void SetTextPtr(Ptr textPtr, long numChars);
```

Use the first `numChars` characters that `textPtr` points to as the text for this static text object.

```
CharsHandle GetTextHandle(void);
```

Return a handle to the text of the static text. The handle contains the actual text, not a copy.

Text characteristics methods

```
void SetFontNumber(short aFontNumber);
```

Set the font for the static text object by font number. This method recalibrates the scales, adjust the bounds, and redraws the static text.

```
void SetFontName(Str255 aFontName);
```

Set the font for the static text object by font name. This method recalibrates the scales, adjust the bounds, and redraws the static text.

```
void SetFontStyle(short aStyle);
```

Set the font style for all the text in the static text object. Style is one of: `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, `extend`

```
void SetFontSize(short aSize);
```

Set the font size for all the text in the static text object to the specified size.

```
void SetTextMode(short aMode);
```

Set the text mode for all the text in the static text object to the specified mode. `AMode` can be one of `srcOr`, `srcXor`, or `srcBic`.

```
void SetAlignment(short anAlignment);
```

Set the text alignment mode for all the text in the static text object to the specified mode. `AnAlignment` can be one of `teJustLeft`, `teJustCenter`, or `teJustRight`.

```
void SetLineSpacing(long aSpacingCmd);
```

Set the line spacing for all the text in the static text object to the specified spacing. `ASpacingCmd` can be one of `cmdSingleSpace`, `cmdlHalfSpace`, `cmdDoubleSpace`.

```
void GetTEFontInfo(FontInfo *macFontInfo);
```

Return the information about the text font, size, and style of the Text Edit record associated with this static text in a Macintosh `FontInfo` record.

Calibrating methods

```
void SetWholeLines(Boolean aWholeLines);
```

If `aWholeLines` is TRUE, this method ill resize the frame so it will display only whole lines. The static text pane is not redrawn.

```
void CalcTERects(void);
```

Recalculate the TE destination and view rectangles from the values of the static text's frame and line width. If the line width was less than or equal to zero, this method uses the right edge of the frame.

```
void ResizeFrame(Rect *delta);
```

Resize the static text's frame when the size of its pane changes. The delta rectangle specifies how each side changes. Positive values mean down and to the right. Negative values mean up and to the left.

```
void AdjustBounds(void);
```

Adjust the bounds of the static text object to match the TE record. You should send this message to the static text object when something happens that might affect the number of lines or the line width.

```
short FindLine(short charPos);
```

Return the line number containing the specified character position. Both line and character numbering start at zero. If the character position is before the start of the text (*a negative* number), this method returns zero. If the character position is beyond the end of the text, this method return the number of the last line.

CStaticText summary

```
struct CStaticText : CPanorama {
    TEHandle  macTE;
    short     lineWidth;
    Boolean   wholeLines;
    long      spacingCmd;

    void      IStaticText(CView *anEnclosure, CBureaucrat *aSupervisor,
                  short aWidth, short aHeight,
                  short aHEncl, short aVEncl,
                  SizingOption aHSizing, SizingOption aVSizing,
                  short aLineWidth);
    void      IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor,
                  Ptr viewData);
    void      IStaticTextX(void);
    void      Dispose(void);

    void      Draw(Rect *area);
    void      Scroll(short hDelta, short vDelta, Boolean redraw);
    void      ScrollToSelection(void);

    void      PrintPage(short pageNum, short pageWidth, short pageHeight);
    void      DonePrinting(void);

    void          SetTextString(Str255 textStr);
    void          SetTextHandle(Handle textHand);
    void          SetTextPtr(Ptr textPtr, long numChars);
    CharsHandle   GetTextHandle(void);

    void      SetFontNumber(short aFontNumber);
    void      SetFontName(Str255 aFontName);
    void      SetFontStyle(short aStyle);
    void      SetFontSize(short aSize);
    void      SetTextMode(short aMode);
    void      SetAlignment(short anAlignment);
    void      SetLineSpacing(long aSpacingCmd);
    void      GetTEFontInfo(FontInfo *macFontInfo);

    void      SetWholeLines(Boolean aWholeLines);
    void      CalcTERects(void);
    void      ResizeFrame(Rect *delta);
    void      AdjustBounds(void);

    short     FindLine(short charPos);
};
```

CSwitchboard 50

Introduction

CSwitchboard implements the class that processes Macintosh Toolbox events and sends the appropriate messages to objects in the THINK Class Library. There is normally only one instance of this class.

Heritage

Superclass	CObject
Subclasses	None

Using CSwitchboard

The single instance of this class handles all the Macintosh Toolbox events and sends messages to objects. The application's `Run()` method repeatedly sends `ProcessEvent()` messages to this object to dispatch messages to the objects that make up your application.

The application initialization method `IApplication()` creates the single instance of CSwitchboard. You should not need to override or access any part of this object. The switchboard is stored in the application's `itsSwitchboard` instance variable.

> **Note:** The only time you should make a subclass of CSwitchboard is to override the `DoOtherEvent()` method if your application handles `appl Evt`, `app2Evt`, or `app3Evt` events.

Variables

`RgnHandle mouseRgn`	Argument for `WaitNextEvent()` to handle cursor adjustment.

Methods

Initialization methods

```
void ISwitchboard(void);
```

Initialize the switchboard. The application's `IApplication()` method *creates the* switchboard and sends it this message.

Mouse methods

```
void DoMouseDown(EventRecord *theEvent);
```

The user pressed the mouse button. This method sends a `DispatchClick()` message to the desktop, and stores the event in the global `gLastMouseDown`.

```
void DoMouseUp(EventRecord *theEvent); /* Mouse button released */
```

The user released the mouse button. This message sends a `DoMouseUp()` message to the last view hit. Since mouse ups always follow a mouse down, it's not important where the mouse came up, but the message must be sent to the same object that handled the mouse down. This method stores the event in the global `gLastMouseUp`.

Key methods

```
void DoKeyEvent(EventRecord *theEvent);
```

The user pressed or released a key. If the user holds down the Command key and presses a key at the same time, this method uses the Toolbox routine `MenuKey()` to find the menu equivalent. If there is a menu equivalent, this method sends a `DoCommand()` message to the gopher. For other key events, this method send a `DoKeyDown()`, `DoKeyUp()`, or `DoAutoKey()` messages to the gopher.

Disk methods

```
void DoDiskEvent(EventRecord *theEvent);
```

This method calls the Toolbox routine `DIBadMount()` to mount a disk. This is the only event that the switchboard handles directly.

Window event methods

```
void DoUpdate(EventRecord *theEvent);
```

This method sends an `Update()` message to the window specified in the event record.

```
void DoActivate(EventRecord *theEvent);
```

This method sends an `Activate()` message to the window specified in the event record.

```
void DoDeactivate(EventRecord *theEvent);
```

This method sends an `Deactivate()` message to the window specified in the event record.

Suspend/Resume methods

```
void DoSuspend(EventRecord *theEvent);
```

The application is about to be switched to the background under MultiFinder. This method sends a `Suspend()` message to your application.

```
void DoResume(EventRecord *theEvent);
```

The application is about to be switched to the foreground under MultiFinder. This method sends a `Resume()` message to your application.

Event processing methods

```
void DoOtherEvent(EventRecord *theEvent);
```

If your application handles `app1Evt`, `app2Evt`, or `app3Evt` events, override this method.

```
void DoIdle(EventRecord *theEvent);
```

This method is invoked during null events. This method sends an `Idle()` message to your application. The application uses its `Idle()` message to perform periodic tasks.

```
void ProcessEvent(void);
```

This method is the main event dispatcher. This method calls one of the Toolbox routines `GetNextEvent()` or `WaitNextEvent()` to get an event. Depending on the event, it sends an appropriate message back to itself to handle the event. Before processing the event, `ProcessEvent()` sends a `DispatchCursor()` message to the application to adjust the cursor.

CSwitchboard summary

```
struct CSwitchboard : CObject {
    RgnHandle           mouseRgn;

    void      ISwitchboard(void);
    void      DoMouseDown(EventRecord*);
    void      DoMouseUp(EventRecord*);
    void      DoKeyEvent(EventRecord*);
    void      DoDiskEvent(EventRecord*);
    void      DoUpdate(EventRecord*);
    void      DoActivate(EventRecord*);
    void      DoDeactivate(EventRecord*);
    void      DoSuspend(EventRecord*);
    void      DoResume(EventRecord*);
    void      DoOtherEvent(EventRecord*);
    void      DoIdle(EventRecord*);
    void      ProcessEvent(void);
};
```

CTask 51

Introduction

CTask is an abstract class for implementing undoable actions.

Heritage

Superclass	CObject
Subclasses	CMouseTask

Using CTask

A task is an abstract class for implementing undoable actions. If you want your application to be able to undo an action, you need to define a task subclass for each action.

You can use a task two ways. You can perform your action, create a task object, store enough information in it for its `Undo()` method to undo the action, and send it in a `Notify()` message to your supervisor (usually the document).

The second way is similar to the first, but you also implement a `Do()` method that performs the action. So you create a task, send it a `Do()` message to perform the action, and then send it in a `Notify()` message to your supervisor. Your `Do()` method stores enough information in the task's instance variables to undo the action.

When you notify a document that you've performed a task, it stores the task in the instance variable `lastTask`. When you choose Undo from the Edit menu, the document's `DoCommand()` method sends an `Undo()` message to that task.

Every task subclass has a string in the `STR# 130` resource used for the wording of the Undo/Redo command. Tasks have an instance variable that is the index of its string in the `STR#` resource. The document's `UpdateUndo()` method takes care of the wording of the Undo/Redo command.

Here's an example. Suppose you've defined a subclass of CTask to change the font in an edit text pane. Before passing the command on to the edit pane's `DoCommand()` method, you create a task and store the current font in an instance variable. After you pass the font command to the edit text pane, you send the task in a `Notify()` message to the document.

Your Undo() method would simply send the font change command to the document. Since the command goes through the regular command chain, your DoCommand() method would create a task to let you undo what you were undoing.

Variables

`short nameIndex;`	Index of the Undo/Redo string in the STR# 130 resource

Methods

Initialization methods

```
void ITask(short aNameIndex);
```

Initialize a task object. ANameIndex is the index of the task's Undo string in the STR# 130 resource. Your subclass's initialization method should call this method in addition to any other initialization it does. If your task subclass allocates memory, you'll also need to implement a Dispose() method to release that memory.

Accessing methods

```
short GetNameIndex(void);
```

Get the task's index. This method is used by the document's UpdateUndo() method.

Action methods

```
void Do(void);
```

Perform a task. The default method does nothing. If you want to use a task to implement an action which is not necessarily undoable, your subclass should override this method. The Undo/Redo mechanism doesn't send Do() messages.

```
void Undo(void);
```

Undo a task. The default method does nothing. Your subclass must store enough information to be able to undo an action. This is the method where you implement the undo.

```
void Redo(void);
```

Redo a task that was undone. The default method sends the task an Undo() message. This method assumes that a redo is the same as undoing the undo. If your application implements Redo differently, you'll need to override this method.

Class resources

STR# 130	List of strings for the wording of the Undo/Redo command. For instance, if you're implementing a "move" action, your string would be "Move". Each task contains the index of its string in this resource.

CTask summary

```
struct CTask : CObject {
    short       nameIndex;

    void        ITask(short aNameIndex);
    short       GetNameIndex(void);

    void        Do(void);
    void        Undo(void);
    void        Redo(void);
};
```

CView
52

Introduction

CView is an abstract class for implementing objects that have a visual representation. All of the objects in the visual hierarchy are descendants of this class.

Heritage

Superclass	CBureaucrat
Subclasses	CDesktop
	CPane
	CWindow

Using CView

CView is an abstract class for implementing objects with a visual representation. In other words, anything you can see on the screen is a descendant of CView. Views respond to visual commands involving the mouse. And because CView is a descendant of CBureaucrat, a view can be one of the links in the chain of command.

All views have an enclosure that specifies its place in the visual hierarchy. Each view can enclose several subviews. The top of the visual hierarchy, the desktop, is the only view that does not have an enclosure. The desktop encloses all the windows, also descendants of CView, in your application. And each window encloses one or mores panes. Panes can enclose other panes.

The desktop handles some visual commands, like mouse clicks, and sends them on to the appropriate window. The switchboard sends window related messages, like updates and activates, directly to a window which sends it to its subviews.

Each view also has a supervisor. The supervisor is the view's boss in the chain of command. The desktop's supervisor is always the application. A window's supervisor is always its director or document. A pane's supervisor is usually its director or document.

The desktop and windows are almost never the first in the chain of command. In other words, they're almost never the gopher. Panes, on the other hand, are frequently made the gopher. For instance, you need to make an edit pane the gopher so it can respond to typing and menu commands.

The standard classes define several subclasses of CView. These are the desktop, windows, and panes. Most of the time, you'll be dealing with panes. As you work with panes and descendants of CPane, keep in mind that all methods that apply to views apply to them as well.

Variables

`GrafPtr macPort;`	Mac drawing port for the view
`CView *itsEnclosure;`	View which totally encloses this one
`CList *itsSubviews;`	Views contained within this view
`Boolean visible;`	Is the view visible?
`Boolean active;`	Is the view active?
`Boolean wantsClicks;`	Does the view handle mouse clicks?

Methods

Construction and destruction methods

```
void IView(CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a view. `AnEnclosure` is the view that completely encloses this view. `ASupervisor` is the bureaucrat that gets command messages when the view can't handle them. Views start out with no port, no subviews, invisible, and inactive. By default, views don't want clicks.

```
void IViewRes(ResType rType, short resID, CView *anEnclosure,
CBureaucrat *aSupervisor);
```

Initialize a view from a resource template. Each subclass of CView overrides this method to use its own resource template. `RTtype` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the same as for `IView()`.

To initialize a view from a resource file, use a `'View'` resource.

```
void IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor, Ptr
viewData);
```

The `IViewRes()` method sends an `IViewTemp()` message to initialize a view from a resource template. All subclasses of CView override this method so they can be initialized from resource templates.

```
void Dispose(void);
```

Dispose of a view. Disposes of all subviews.

Accessing methods

```
Boolean IsVisible(void);
```
Return TRUE if the view is visible.

```
Boolean IsActive(void);
```
Return TRUE if the view is active.

```
Boolean ReallyVisible(void);
```
Return TRUE if the view is visible and if its enclosure is `ReallyVisible()`.

```
GrafPtr GetMacPort(void);
```
Get the GrafPort associated with this view.

```
void GetOrigin(long *theHOrigin, long *theVOrigin);
```
Get the origin of the view (he top, left corner). The default method returns (0, 0).

```
void GetFrame(Rect *theFrame);
```
Get the frame of the view. The default method does nothing. View subclasses (like Pane) must override this message.

```
void GetInterior(Rect *theInterior);
```
Get the interior of the view. The default method returns what `GetFrame()` returns. If the interior of a particular subclass is not the same as the frame, the class must override this method.

```
void GetAperture(Rect *theAperture);
```
Get the aperture of the view. (The aperture is the visible portion of a view.) The default method does nothing. Subclasses must override this method.

```
Boolean Contains(Point thePoint);
```
Return TRUE if the view contains `thePoint`. The default method always returns FALSE.

```
void SetWantsClicks(Boolean aWantsClicks);
```
If `aWantsClicks` is TRUE, the view will report clicks in itself. By default, views don't want clicks.

Appearance methods

```
void Show(void);
```
Make a view visible.

```
void Hide(void);
```
Hide a view. If the view is the gopher, make its enclosure the gopher.

```
void Activate(void);
```
Make a view active. Activate all the subviews as well.

```
void Deactivate(void);
```
Make a view inactive. Deactivate all the subviews as well.

Mouse methods

```
void DispatchClick(EventRecord *macEvent);
```
Find out which subview got the click and send it a `DispatchClick()` message. If there are no subviews, the click is for this view, so send it a `DoClick()` message. There where field of the `macEvent` record is in local coordinates. You should not override this method.

```
void DoClick(Point hitPt, short modifierKeys, long when);
```
The user has clicked in this view. The hitPt is in local coordinates. Your subclass must override this method.

```
Boolean HitSamePart(Point pointA, Point pointB);
```
Check whether two points hit the same part of the view. The default method always returns TRUE. Subclasses that override this method should decide what constitutes a part and how close is close.

```
void DoMouseUp(EventRecord *macEvent);
```
The mouse went up in a view. Default method does nothing.

```
void TrackMouse(CMouseTask *theTask, Point startPt, Rect
*pinRect);
```
Track the mouse in this view. `TheTask` is a mouse task that you create. `StartPt` is the starting point in frame coordinates. `PinRect` is a constraining rectangle in frame coordinates. This message is usually sent from a `DoClick()` method.

The default method does several things:

- Sends a `Prepare()` message to the view
- Sends a `BeginTracking()` message to theTask
- Sends a `KeepTracking()` message to `theTask` as long as the mouse is down. The current point is constrained to pinRect. (Your task subclass must override the `KeepTracking()` method.)
- Sends an `EndTracking()` message to `theTask`. (Your task subclass must override this method as well.)

To learn more about mouse tracking and mouse tasks, see the class CMouseTask.

Cursor methods

`void DispatchCursor(Point where, RgnHandle mouseRgn);`

Find which view the cursor is in. If the cursor is in this view, send it an `AdjustCursor()` message. If the cursor is in a subview, send it a `DispatchCursor()` message. The point `where` is given in window coordinates.

`void AdjustCursor(Point where, RgnHandle mouseRgn);`

Adjust the cursor. Your view gets an `AdjustCursor()` message whenever the cursor moves into it. The default method sets the cursor to an arrow. Your subclass should override this method to set the cursor to whatever is appropriate for the view. See the `AdjustCursor()` method for CEditText for an example.

If you want only one cursor within a view, you can ignore the two parameters to this method. If you want to use different cursor shapes within a pane, you need to know what the two parameters are for.

> **Note:** It's very unlikely that you'll want multiple cursors for a view that's not a pane. The CPane class inherits this method.

The `where` parameter tells you where the cursor is in window coordinates. You can use the `WindToFrame()` pane method to convert it to frame coordinates. The `mouseRgn` parameter is the region in which the cursor shape stays the same. In other words, your pane will not get an `AdjustCursor()` message again until the cursor leaves this region. The `mouseRgn` is specified in global coordinates.

Suppose that you're displaying a map of the United States in a pane, and that you want the cursor to be a star when it's over the state of Texas. You use the `where` parameter, converted from window coordinates to frame coordinates, to determine that you're in Texas, so you change the cursor to a star.

Now, when you leave Texas, you want to set the cursor shape to something else. Since your pane gets `AdjustCursor()` messages only when the cursor leaves the `mouseRgn`, you have to change the `mouseRgn`. You create a region with the shape of the state of Texas, convert this region to global coordinates, and make it the `mouseRgn`.

> **Note:** The mouse region should be the intersection of the original mouse region and the one you're specifying. This way, you preserve any clipping boundaries that the original mouse region had accounted for.

This is what the `AdjustCursor()` method for this example might look like:

```
void MapPane::AdjustCursor(Point where, RgnHandle mouseRgn)
{
    Point       locWhere;
    RgnHandle   TexasRgn = NewRgn();
```

```
    Rect        TexasRect;
    Rect        tempRect;

    locWhere = where;
    WindToFrame(&locWhere);

        /*  If we're not in Texas, use the default */
    if (!PtInTexas(locWhere)) {
        inherited::AdjustCursor(where, mouseRgn);
        return ;
    }

        /* Set the star cursor */
    SetCursor(star);

        /* calculate the mouse region in frame coords */
    OpenRgn();
    DrawTexas();
    CloseRgn(TexasRgn);

        /* convert it to global coords */
    TexasRect= (**TexasRgn).rgnBBox;
    tempRect = TexasRect;
    FrameToGlobalR(&tempRect);
    OffsetRgn(TexasRgn, tempRect.left - TexasRect.left
                        tempRect.top - TexasRect.top);

        /* set the mouse region */
    SectRgn(mouseRgn, TexasRgn, mouseRgn);
}
```

Subview Management methods

`void AddSubview(CView *theSubview);`

Add `theSubview` to the view's `itsSubviews` list. You should not use or override this method.

`void RemoveSubview(CView *theSubview);`

Remove `theSubview` from the `itsSubviews` list.

`CView* FindSubview(Point hitPt);`

Find the subview that contains `hitPt`. `HitPt` is in window coordinates. This method finds the topmost subview. You should not override thie method.

```
void SubpaneLocation(short hEncl, short vEncl, long *hLocation,
long *vLocation);
```

Return the position of a subview at `hEncl`, `vEncl` in window coordinates (in `hLocation` and `vLocation`)

```
void Prepare(void);
```

Prepare to draw or to do something after a click. The default method does nothing. View classes must override this method.

```
void FrameToGlobalR(Rect *theRect);
```

Convert `theRect` from frame coordinates to global coordinates.

Class resources

The `IViewRes()` method lets you initialize any descendant of CView with a resource template.

Resource	Class
`Bord`	CBorder
`Pane`	CPane
`Pano`	CPanorama
`PctP`	CPicture
`ScPn`	CScollPane
`StTx`	CStaticText, CEditText
`View`	CView

You can use ResEdit to create the resource templates for each class.

> **Note:** Your THINK C package includes a file called `TCL TMPLs` that contains a set of `TMPL` resources you can install into ResEdit. These `TMPLs` let you create and edit the resource above easily. See the instructions in "Installing the THINK Class Library" in Chapter 16 to learn how to install these `TMPLs` into ResEdit.

CView summary

```
struct CView : CBureaucrat {
    GrafPtr         macPort;
    CView           *itsEnclosure;
    CList           *itsSubviews;
    Boolean         visible;
    Boolean         active;
    Boolean         wantsClicks;

    void            IView(CView *anEnclosure, CBureaucrat *aSupervisor);
    void            IViewRes(ResType rType, short resID,
                             CView *anEnclosure, CBureaucrat *aSupervisor);
    void            IViewTemp(CView *anEnclosure, CBureaucrat *aSupervisor,
                              Ptr viewData);
    void            Dispose(void);

    Boolean         IsVisible(void);
    Boolean         IsActive(void);
    Boolean         ReallyVisible(void);
    GrafPtr         GetMacPort(void);
    void            GetOrigin(long *theHOrigin, long *theVOrigin);
    void            GetFrame(Rect *theFrame);
    void            GetInterior(Rect *theInterior);
    void            GetAperture(Rect *theAperture);
    Boolean         Contains(Point thePoint);
    void            SetWantsClicks(Boolean aWantsClicks);

    void            Show(void);
    void            Hide(void);
    void            Activate(void);
    void            Deactivate(void);

    void            DispatchClick(EventRecord *macEvent);
    void            DoClick(Point hitPt, short modifierKeys, long when);
    Boolean         HitSamePart(Point pointA, Point pointB);
    void            DoMouseUp(EventRecord *macEvent);
    void            TrackMouse(CMouseTask *theTask, Point startPt, Rect *pinRect);

    void            DispatchCursor(Point where, RgnHandle mouseRgn);
    void            AdjustCursor(Point where, RgnHandle mouseRgn);

    void            AddSubview(CView *theSubview);
    void            RemoveSubview(CView *theSubview);
    CView*          FindSubview(Point hitPt);
    void            SubpaneLocation(short hEncl, short vEncl,
                                    long *hLocation, long *vLocation);
    void            Prepare(void);
    void            FrameToGlobalR(Rect *theRect);
};
```

CWindow
53

Introduction

CWindow implements a class necessary to manipulate a Macintosh window.

Heritage

Superclass	CView
Subclasses	None

Using CWindow

Virtually every application you write on the Macintosh uses windows. The CWindow class implements methods to manipulate Macintosh windows. In the THINK Class Library, window objects are merely visual entities. They're not designed to interact directly with the application. The class CDirector manages communication between a window and the application. The CDocument class, a subclass of CDirector, manages communication between the application, a window, and a file.

Objects of class CWindow usually belong to directors. Under normal circumstances, you should not need to define a subclass of CWindow or to manipulate one directly. The only thing you really need to do to a window is create it and set its options.

The Macintosh windows associated with CWindow objects have a window kind of `OBJ_WINDOW_KIND`. If your application uses windows that are not associated with a CWindow object— from a utility library, for instance— you should override the CDesktop methods `DispatchClick()` and `DispatchCursor()` and the CSwitchboard methods `DoUpdate()`, `DoActivate()`, and `DoDeactivate()` to check the window kind of the Macintosh windows they deal with.

Variables

The enclosure for all windows must be `gDesktop`.

Global variable

`CDesktop *gDesktop;`	The global desktop which acts as the top of the visual hierarchy and the enclosure for all windows.

Local variables

`Rect sizeRect;`	Minimum and maximum size of the window
`Boolean floating;`	Is this a floating window?
`Boolean actClick;`	Process mouse click which activates window?
`Point location;`	Current window location, used when "hiding" a window while suspended

Methods

Construction and destruction methods

```
void IWindow(short WINDid, Boolean aFloating, CView *anEnclosure,
CBureaucrat *aSupervisor);
```

Initialize a window object. `WINDid` is the id of the `WIND` resource. If `aFloating` is TRUE, the window is a floating window. In this case, `anEnclosure` must refer to a floating window desktop.

`AnEnclosure` is the view the window belongs to. A window's enclosure should always be `gDesktop`. `ASupervisor` is the window's supervisor.Usually a window's supervisor is the director it belongs to. `IWindow()` sends the window a `MakeMacWindow()` message to actually allocate space in memory for the window.

```
void Dispose(void);
```

Dispose of the window and all of its subpanes. This method also removes the window from the desktop.

```
void MakeMacWindow(short WINDid);
```

Create a Macintosh window. The default method uses `WIND` resource and lets the Window Manager allocate memory itself. If you want to create your windows differently, override this method. Bear in mind that floating windows should be in front (-1L) and that non-floating windows should be behind.

```
void Close(void);
```

Close a window. A window object gets this message as a result of a click in the close box. The default method sends the window's supervisor a `CloseWind()` message.

Accessing methods

```
void GetFrame(Rect *theFrame);
```

Get the frame of a window. The frame of the window includes the one pixel border around the window. The top, left of `theFrame` will always be (-1,-1)

```
void GetInterior(Rect *theInterior);
```

Get the interior of a window. The interior of the window is the same as its `portRect` and does not include any of the border pixels. The top left of `theInterior` will always be (0, 0).

```
void GetAperture(Rect *theAperture);
```

Return the drawable area of the window. For windows, this is the entire window. The top, left of `theAperture` will always be (0,0).

```
Boolean IsFloating(void);
```

Return true if this is a floating window.

```
void SetTitle(Str255 theTitle);
```

Set the window's title.

```
void GetTitle(Str255 theTitle);
```

Get the window's title

```
void SetActClick(Boolean anActClick);
```

If `anActClick` is TRUE, the window will process the mouse click that activates it.

```
Boolean WantsActClick(void);
```

Return TRUE if the window processes the mouse click that activates it.

```
Boolean Contains(Point thePoint);
```

Return TRUE if `thePoint` is inside the window. `ThePoint` is in global coordinates.

```
void SetSizeRect(Rect *aSizeRect);
```

Set the minimum and maximum size for the window. `ASizeRect`'s top,left are the minimum height and width. `ASizeRect`'s bottom,right are the maximum.

```
void SetStdState(Rect *aStdState);
```

Set the standard state of a window. The standard state is the size and location of the window when it's zoomed out. Before you send this message to a window, make sure that

the Macintosh window associated with it supports zooming. If it does, the `spareFlag` in the window record is TRUE.

Appearance methods

```
void Show(void);
```

Show a window. Also sends its enclosure (the desktop) a `ShowWind()` message.

```
void Hide(void);
```

Hide a window Also sends its enclosure (the desktop) a `HideWind()` message.

```
void Activate(void);
```

Activate a window and all of its panes. Also sends its supervisor (a director) an `ActivateWind()` message.

```
void Deactivate(void);
```

Deactivate a window and all of its panes. Also sends its supervisor (a director) a `DeactivateWind()` message.

```
void Select(void);
```

Select a window by bringing it to the front and making it active. Sends its enclosure (the desktop) a `SelectWind()` message.

```
void ShowResume(void);
```

Make a window visible when an application resumes. If you hide your windows when your application is suspended, send it this message in its director's `Resume()` method.

```
void HideSuspend(void);
```

Hide a window when an application is suspended. To hide a window when the application is being suspended, send the window this message in its director's `Suspend()` method. The `HideSuspend()` method doesn't actually hide the window. Instead, it moves *it out* of the visible range so the front-to-back ordering of the windows doesn't change.

```
void ShowOrHide(Boolean showFlag);
```

Make a window visible or invisible without generating any update or activate events.

Size and location methods

```
void Drag(EventRecord *macEvent);
```

Drag a window. This method is invoked when the user clicks in a window's drag region. The default method sends a `DragWind()` message to the desktop.

```
void Resize(EventRecord *macEvent);
```

Resize a window. This method is invoked when you click and drag the window's grow region. After resizing, this method sends a `ChangeSize()` message to the window.

```
void Zoom(short direction);
```

Zoom a window. This method is invoked when the user clicks in a window's zoom box. Direction is either `inZoomIn` or `inZoomOut`. After changing the size of the window, this method sends an `AdjustToEnclosure()` message to its panes.

```
void Move(short hGlobal, short vGlobal);
```

Move the window to the specified location. Note that the new position refers to the top left corner of the window's content region. The title bar will be above the specified coordinates.

```
void ChangeSize(short width, short height);
```

Change the size of the window to the specified width and height. After changing the window size, this method sends an `AdjustToEnclosure()` message to its panes. This method respects the maximum and minimum window sizes you set with SetSizeRect().

```
void MoveOffScreen(void);
```

Move the window out of the visible area of the desktop.

Drawing methods

```
void Update(void);
```

Update the window. This method is invoked when the Switchboard processes an update event. This method sends a `Prepare()` message to the window and then sends a `Draw()` message to each of the window's panes. You should not override this method.

```
void Prepare(void);
```

Set the port of the window and prepare for drawing.

Mouse methods

```
void DispatchClick(EventRecord *macEvent);
```

Find a pane to handle the mouse click. You should not use or override this method. If you want to handle a click in a particular window subclass, don't override this method. Override the `DoClick()` method (inherited from CView) instead.

```
void DispatchCursor(Point where, RgnHandle mouseRgn);
```

Find a pane that handles `AdjustCursor()` messages. You should not use or override this method.

Conversion methods

`FrameToGlobalR(Rect *theRect);`

Convert `theRect` from frame to global coordinates.

CWindow summary

```
struct CWindow : CView {
   Rect      sizeRect;
   Boolean   floating;
   Boolean   actClick;
   Point     location;

   void      IWindow(short WINDid, Boolean aFloating,
                     CView *anEnclosure, CBureaucrat *aSupervisor);
   void      Dispose(void);
   void      MakeMacWindow(short WINDid);
   void      Close(void);

   void      GetFrame(Rect *theFrame);
   void      GetInterior(Rect *theInterior);
   void      GetAperture(Rect *theAperture);
   Boolean   IsFloating(void);
   void      SetTitle(Str255 theTitle);
   void      GetTitle(Str255 theTitle);
   void      SetActClick(Boolean anActClick);
   Boolean   WantsActClick(void);
   Boolean   Contains(Point thePoint);
   void      SetSizeRect(Rect *aSizeRect);

   void      Show(void);
   void      Hide(void);
   void      Activate(void);
   void      Deactivate(void);
   void      Select(void);
   void      ShowResume(void);
   void      HideSuspend(void);
   void      ShowOrHide(Boolean showFlag);

   void      Drag(EventRecord *macEvent);
   void      Resize(EventRecord *macEvent);
   void      Zoom(short direction);
   void      Move(short hGlobal, short vGlobal);
   void      ChangeSize(short width, short height);
   void      MoveOffScreen(void);

   void      Update(void);
   void      Prepare(void);

   void      DispatchClick(EventRecord *macEvent);
   void      DispatchCursor(Point where, RgnHandle mouseRgn);

   void      FrameToGlobalR(Rect *theRect);
};
```

Global Variables
54

Introduction

This chapter describes the global variables in the THINK Class Library.

Global objects

These globals hold pointers to the only instances of some classes. Most of them are initialized in the application initialization routines. The only variable you should set yourself during the course of your application is `gGopher` to set the first bureaucrat in the chain of command.

`CApplication *gApplication;` — The global application object. You should set this variable to an instance of your application class in your main program. See CApplication for an example.

`CDesktop *gDesktop;` — The desktop. This variable holds the only instance of CDesktop. It's initialized in the application method `MakeDesktop()`. The desktop is the enclosure for all the windows in your application. It is the top of the visual hierarchy.

`CBartender *gBartender;` — The bartender. This variable holds the only instance of CBartender. It's initialized in the application method `SetUpMenus()`. The bartender converts menu selections into command numbers and handles most menu operations.

`CClipboard *gClipboard;`	The clipboard. This variable holds the only instance of CClipboard. It's initialized in the application method `MakeClipboard()`. The clipboard is where data is Cut and Copied to and Pasted from.
`CBureaucrat *gGopher;`	The gopher is the first bureaucrat to get commands. If the gopher can't handle the command, it passes control to its supervisor. Initially, the gopher is set to the application (`gApplication`). When a document becomes active, the gopher points to the document. A document can set the gopher to point to one of its panes.
`CError *gError;`	The global error handler. This variable is initialized in `IApplication()`.
`CDecorator *gDecorator;`	The window dresser. This variable holds the only instance of CDecorator. It's initialized in the application method `MakeDecorator()`. The decorator takes care of arranging windows on the screen.

MultiFinder globals

You can use these globals to find out whether your application is in the background and whether `WaitNextEvent()` is implemented. You can set `gSleepTime` to zero if you want to force an idle event.

`long gSleepTime;`	The switchboard uses this value to pass to `WaitNextEvent()`. It is the maximum time (in ticks) between events. `Perform()` methods in CChore subclasses and `Dawdle()` methods in CBureaucrat subclasses can change this value indirectly to force an idle event.
`Boolean gHasWNE;`	TRUE if `WaitNextEvent()` is implemented.

`Boolean gInBackground;`	TRUE if the application is in the background.

Mouse click globals

The THINK Class Library uses these globals to count mouse clicks. The only variable you'll need to use is `gClicks`.

`EventRecord gLastMouseDown;`	Event record of the last mouse-down event.
`EventRecord gLastMouseUp;`	Event record of the last mouse-up event.
`CView *gLastViewHit;`	The last view the mouse went down in.
`short gClicks;`	Click counter. This variables counts multiple clicks. Its value is 1 for a single click, 2 for a double click, 3 for a triple click, etc. To be considered a multiple click the mouse must go down in the same view as the last mouse down within the amount of time specified by `GetDblTime()` and the view method `HitSamePart()` must return TRUE.

Cursors

These cursor handles are initialized in `IApplication()`. You can use them as arguments to `SetCursor()`. Remember that these are handles, so you'll have to call `SetCursor()` like this:

```
SetCursor(*gWatchCursor);   /* Dereference the handle */
```

`CursHandle gIBeamCursor;`	I-beam for text views.
`CursHandle gWatchCursor;`	Watch cursor for waiting.

Utility globals

`OSType gSignature;`

The signature of your application. You should initialize this variable in your `SetUpFileParameters()` application method. Use this variable whenever your application needs to create a file.

`RgnHandle gUtilRgn;`

Utility region. This region is initialized with `NewRgn()` in `IApplication()`. You can use it wherever you need a region, but you should not rely on it being the same across calls to different methods.

THINK C

PART FIVE

Reference

55 THINK C Menus
56 Debugger Menus
57 Language Reference

THINK C Menus
55

Introduction

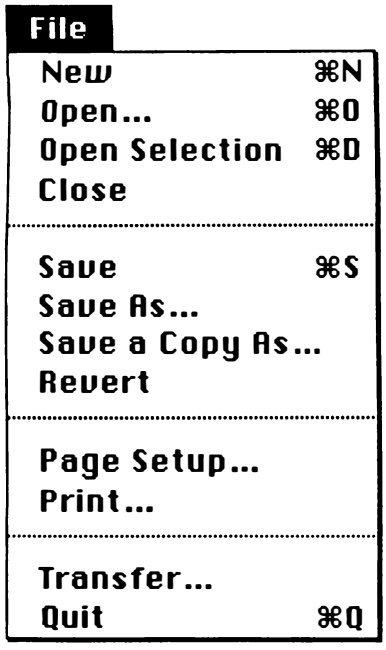
This chapter describes each of the THINK C menu commands. It is organized by menu, from left to right along the menu bar. Within each menu, commands are described in the order in which they appear in the menu.

The Menu

About THINK C...

This command tells you what version of THINK C you're using. When you choose this command, you'll see a star field. Click the mouse to see the version of THINK C. Click the mouse two more times to end the display.

The File Menu



Use the **File** menu commands to work with files that you open and edit with the THINK C text editor. This menu also has the commands that let you launch other applications and that let you quit THINK C.

New

This command opens a new Untitled window. You must save this file with a .c extension if you plan to compile it or add it to the project. However, you can use the **Check Syntax** command on the **Source** menu to compile it without adding it to the project.

Open...

This command displays a dialog box that lets you select from existing files on the disk and open one for editing. When you first begin a project, one way to add a file is to open it with this command, then compile it or add it with the **Add** command. (The **Add...** command lets you add multiple files without compiling or opening the files. Once files have been added to the project, they can be opened by double-clicking on the file name in the project window.)

You can open multiple files and the edit windows will stack up with the titles showing, so you can easily click on any window to bring it to the front.

If you open too many files at once, you may have to close some windows to free up memory when you compile.

Open Selection

This command lets you open an #included header file simply by selecting its name in the current program text. You don't need to select the .h extension (or full path name for HFS files). The selection will automatically be extended to the right to include it as long as the file name itself is selected.

Close

This command is the same as clicking on the active window's close box. If you try to close an edit window, and the file has been modified since it was last saved, a dialog box will ask you if you want to save the changes, discard them, or cancel the **Close** command. To close the project window, use the **Close Project** command in the **Project** menu.

If the **Confirm Saves** option in the **Options...** dialog box is off, THINK C will save changed files without asking.

Save

This command saves the file in the active edit window to disk. If the file is currently untitled, a dialog box will ask you to name the file.

Note that an updated file can be compiled and added to the project without being saved, as long as it has been saved at least once and given a name with a .c extension.

Save As...

This command lets you save the current file under another name. If you have made edits in the current session, they will be saved under the new name. The original file will remain unchanged, and as you continue editing, you will be editing the new file. This feature is useful for switching to a new version of a file, leaving the old file as a backup.

Save As... tries to preserve the tie between the file you are editing and its entry in the project window. If the file appears in the project window, and the name you want to save it as has a .c extension, and if the new name doesn't already appear in the project window, then the entry for the file in the project window is changed to match the new file name. Use **Save a Copy As...** if you don't want this to happen.

Save a Copy As...

Unlike **Save As...**, this command does not affect the status of the file currently being edited; it simply snapshots it to another file. This is a good way to make backups without finding yourself editing the backup!

Neither **Save As...** nor **Save a Copy As...** will let you save into a file already open in an edit window.

Revert

This command restores the last-saved version of the current file, and discards any edits made in the current session.

Page Setup...

This command displays the standard Page Setup dialog that lets you specify the size of the paper you're printing on, and whether the file should be printed upright on the page (tall orientation) or sideways (wide orientation). See your Macintosh owner's manual for details.

Print...

This command lets you print the current file or the Link Errors window. The **Print...** dialog box lets you set the page range among other options. You can also choose the "Print Pages in Reverse Order" option. When you press the OK button, your file will begin to print. Each page of the file has a header showing the name of the file and the last modification date. The thumb of the vertical scroll bar moves from top to bottom (when printing in forward order), and from bottom to top (when printing in reverse order), to show you the printing progress. To cancel printing, press Command-Period.

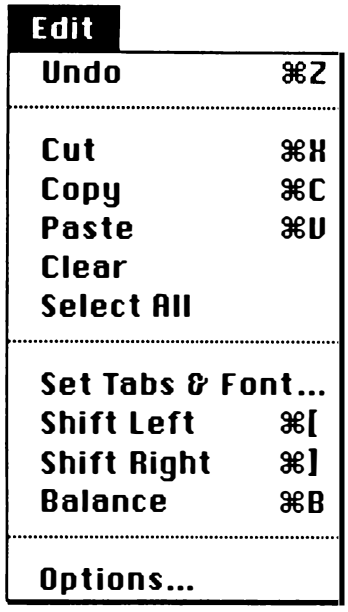
Transfer...

This command lets you launch another application without first returning to the Finder. When you're running under MultiFinder, this command launches applications without quitting THINK C.

Quit

This command exits THINK C and returns to the Finder.

The Edit Menu



The **Edit** menu contains the standard Macintosh editing commands (Cut, Copy, Paste) as well as other commands that let you format your THINK C source files. The **Edit** menu also contains the **Options...** command that lets you set several THINK C options so the environment suits your personal needs.

Undo

The Undo command reverses the last edit operation. The actual name of this command changes to let you know exactly what operation you'll be undoing. After a Paste, for instance, the name of this command changes to Undo Paste. Once you've undone something, the name of this command changes to Redo.

If there isn't anything to Undo, this command will be dimmed. If the operation to undo doesn't belong to the frontmost window, the name of the command will indicate that there is something to do, but the command will be dimmed.

You can't undo a **Replace All**, **Revert**, or a **Set Tabs & Font...** command.

Cut

This command removes selected text and places it in the Clipboard. It replaces the current contents of the Clipboard (if there are any). Use the Paste command to insert text from the Clipboard into your file at the insertion point.

Copy

This command copies the selected text and places it in the Clipboard. The copy can now be pasted somewhere else using the **Paste** command .

Paste

This command copies the contents of the Clipboard into the file being edited at the insertion point. If text is currently selected, it is replaced.

Clear

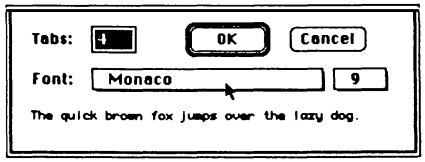
This command clears the selected text. The selection is not placed on the Clipboard. The Clear key on your keyboard has the same effect as the Clear command.

Select All

This command selects all the text in the current edit window.

Set Tabs & Font...

This command lets you change the tab stops and the font used by the THINK C editor. You can select different tab stops and fonts for each edit window. When you choose this command you'll see a dialog box like this one:



Type a number to set the number of spaces per tab. To change the font, click on the font name. You'll see a pop up menu with the names of the fonts in your System file. Click on the font size to see a pop up menu of the sizes for the font.

If you are using a proportionally spaced font like New York or Geneva, the THINK C editor uses the width of the letter m to figure out the width of a tab.

When you change the font or tab settings, the editor adds EFNT and ETAB resources to your text files to record the new settings. Other text editors use these resources as well.

Shift Left

This command shifts a selected range of lines to the left. It deletes the first character of each line in the selected range, as long as the line begins with a tab.

Shift Right

This command shifts a selected range of lines to the right. It inserts a tab at the start of each line in the selected range.

Balance

This command extends the current selection in both directions until it encloses the smallest surrounding balanced text enclosed in parentheses (), brackets [], or braces {}. Successive invocations select larger sequences of text.

This is a quick way to check whether all your function definitions are properly balanced. Start at the beginning of a file and search for the first left brace ("{"). Then use Balance and Find Again commands repeatedly until you get to the end of the file.

Balance is a textual operation. It doesn't know about comments or strings, so if you have a lone brace, bracket, or parentheses in a comment, it will try to find a match for it.

Options...

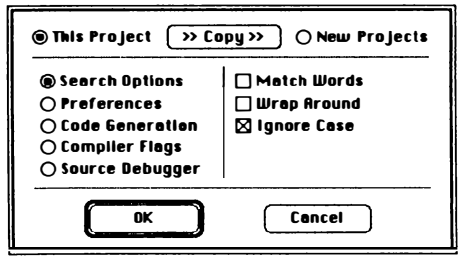
The **Options...** command opens a dialog box that lets you set five groups of options: search options in the editor, project preferences, code generation options, compiler options, and debugger options.

You can set the options for the current project, or you can set the defaults that THINK C will use when you create a new project. Use the Copy button at the top of the dialog box to copy the THINK C defaults to the current project, or, if you want the options you've set for a particular project to be the THINK C options. Note that even though only one section of the options shows up in the dialog at one time, the Copy button copies *all* of the options, even the ones in the sections you can't see.

When you click on the OK button, the changes for all sections of the dialog are saved.

SEARCH OPTIONS

The Search Options section of the **Options...** dialog lets you set the defaults used in the **Find...** dialog. (When you set the same options in the **Find...** dialog, they apply only for the current session.)



Match Words — When this option is set, the editor's search and replace functions will work on whole words only. For example, a search for "string" will not match the word "strings." This option is off by default.

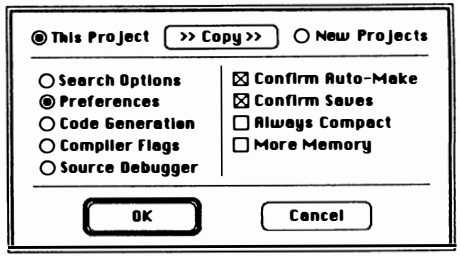
Wrap Around — When this option is set, the editor's search and replace functions will search the entire file, rather than from the current position to the end of the file. When the end of the file is reached, the search "wraps around" to the beginning of the file and continues. This option is off by default.

Ignore Case — When this option is set, the editor's search and replace functions will disregard case when performing a search. A search

string will match either upper or lower case. This option is on by default.

PREFERENCES

The Preferences section lets you specify how THINK C behaves when it rebuilds projects, closes files, closes projects, and how it deals with memory.



Confirm Auto-Make When this option is off, THINK C does not display the "Bring project up to date?" dialog when you choose the **Run** or any of the **Build...** commands. THINK C assumes you would have answered Yes. This option is on by default.

Confirm Saves When this option is off, THINK C automatically saves changes to a file that you have modified without asking if you are sure you want to do so. This is a dangerous option to turn off, since it protects you from inadvertently replacing previous versions of your files with newly modified versions. It is, however, very convenient when you want to do program development in quick, incremental steps. This option is on by default.

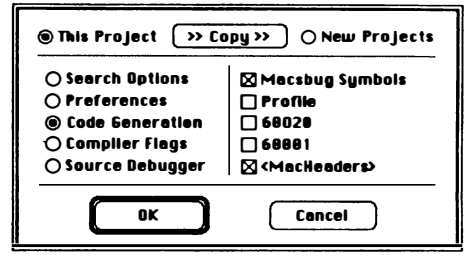
Always Compact In order to achieve its remarkable speed, THINK C pre-allocates space in the project for anticipated requirements, and does not necessarily free up the space when an item is deleted. As much as 20% of an uncompacted project document can contain unused space. If you turn this option on, the project document will be compacted when you **Close** the project, **Transfer...**, or **Quit** (but not when you **Run**). Compaction may be time-consuming , so you will normally want this this option on only when disk space is at a premium. The amount of space freed will vary. To compact a project without setting this option, use the **Close & Compact Project** command in the **Project** menu. This option is off by default.

More Memory When this options is on, THINK C tries to find more memory during compilation by discarding certain data structures. This incurs some additional disk activity, since (1) the data structures will need to be read back in when compilation is complete, and (2) if they are "dirty", the data structures must be written out prior to compilation in case the compiler runs out

of memory. Leave this option unchecked unless you are developing a large project on a small machine which keeps running out of memory.

CODE GENERATION

The Code Generation section lets you control how THINK C generates code.



Macsbug Symbols When the Macsbug Symbols options is set, THINK C generates symbols for assembly language level debuggers such as Macsbug or TMON. THINK C generates symbols only for functions that have stack frames, so functions without arguments and local variables don't get symbols. Be aware that while Macsbug symbols are useful for debugging, they add 8 bytes to every procedure. This option is on by default.

Profile When the Profile option is set, THINK C generates calls to code profiler routines. The code profiler collects timing statistics about your functions. See Appendix A to learn more about the code profiler. This option is off by default.

68020 If the 68020 option is checked, THINK C uses the MC68020 instructions for bitfield operations and long word multiplication, division, and modulo operations. Also, the inline assembler accepts all MC68020 instructions and addressing modes for assembly in an `asm { ... }` construct.

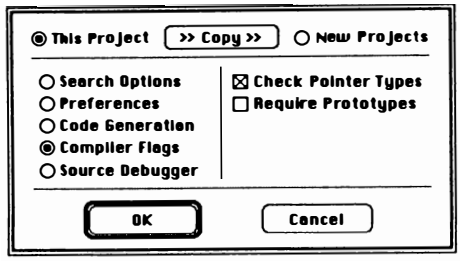
68881 If the 68881 option is checked, THINK C generates code for the floating point coprocessor. Up to five local variables of type `double` may be declared `register` and will be placed into 68881 registers. When this option is on, the size of double variables is 96 bits. Since SANE expects 80 bit doubles, you'll need to convert from one format to the other. The `float` (32 bits) and `short double` (64 bits) types are identical for SANE and for the 68881.

<MacHeaders> When the <MacHeaders> option is on, THINK C will automatically include the `MacHeaders` file for every file in your project. The `MacHeaders` file contains the declarations for the

most common Macintosh Toolbox types, functions, and low memory globals. Since these declarations are in binary form, compilation is faster than if you included the header files manually. It doesn't hurt to include header files like `QuickDraw.h`, but compilation will be a bit slower. If you want to use your own precompiled headers, make sure this option is turned off. THINK C allows only one precompiled header per source file. To learn more about precompiled headers, see the "Precompiled Headers" section in Chapter 10.

COMPILER FLAGS

The Compiler Flags section of the Options... section lets you specify how the THINK C compiler interprets your source files.

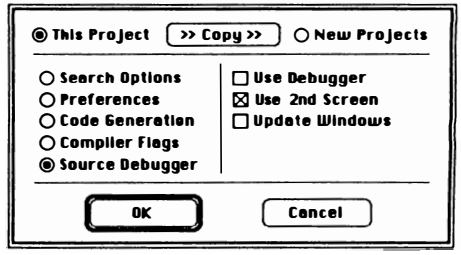


Check Pointer Types When the Check Pointer Types option is on, THINK C makes sure that pointer types match when you assign one pointer to another or when you do pointer arithmetic. If this option is off, THINK C treats all pointers as equivalent types, and won't display the "pointer types do not match" error message. When subtracting two pointers, however, the two types must be pointers to objects of the same size.

Require Prototypes When the Require Prototypes option is on, THINK C forces very strict type checking: You can't use or define a function unless it has a prototype. (Macintosh Toolbox routines don't need prototypes even if this option is on.) New-style function definitions do *not* count as prototypes. Read the "Function Prototypes" section in Chapter 10 to learn about function prototypes. By default, the option is not set.

SOURCE DEBUGGER

The Source Debugger section lets you specify whether to use the source level debugger, where the debugger windows will show up, and how your application's windows will be updated.

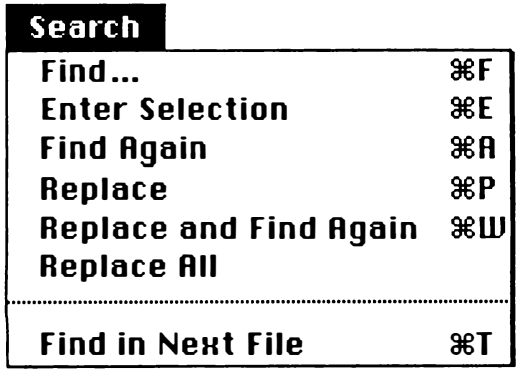


Use Debugger When this option is checked, THINK C launches the source level debugger when you run your project. Checking this option is the same choosing the **Use Debugger** command in the **Project** menu.

Use 2nd Screen When this options is on, and you're running on a Macintosh with more than one screen, THINK C will display the source debugger windows in the second screen.

Update Windows When this option is on, the debugger tries to update your windows for you when your project is stopped. Without this option checked, your program must wait until control comes back so it can handle updates itself. Since it cannot do so until it gets back to its event loop, an update may remain pending for some time. This option is especially useful when you're trying to step through code that draws in a window. This option uses memory to save your window's image. If there isn't enough memory, the debugger won't be able to perform automatic updates. If you have a large or color screen, you might want to increase the debugger's partition size. See Chapter 11 for details.

The Search Menu

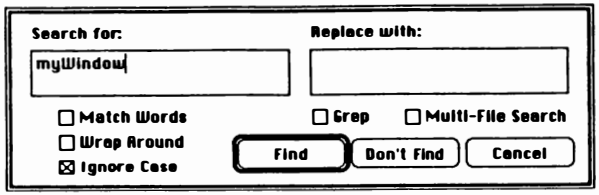


The commands in the **Search** menu let you find and replace strings in your source files. THINK C has extensive search and replace functions including multi-file searching and pattern matching searching. To learn the details of pattern searching, see Chapter 8.

Find...

This command lets you specify a string to search for. If the string is found, it is highlighted. If it is not found, the editor simply beeps.

At the start of an editing session, only the **Find...** command is active. The dialog box that appears in response to this command lets you specify a string to search for, as well as an optional replacement string.



You can also set search options in this dialog box. Note that you can also set these options in the Search Options section of the **Options...** dialog. When you set the options in the Find... dialog, though, they apply only to the current session. If you want to set the defaults permanently for the project you're working on or for new projects you create, use the **Options...** dialog.

Match Words When this option is set, the editor's search and replace functions will work on whole words only. For example, a search for "string" will not match the word "strings". This option is off by default.

Wrap Around When this option is set, the editor's search and replace functions will search the entire file, rather than from the current position to the end of the file. When the end of the file is reached, the search "wraps around" to the beginning of the file and continues. This option is off by default.

Ignore Case When this option is set, the editor's search and replace functions will disregard case when performing a search. A search string will match either upper or lower case. This option is set by default.

In addition to the Find button ("Go ahead with the search") and the Cancel button ("Pretend I never invoked this command"), there is a Don't Find button in the dialog box. Pressing this button causes the editor to accept the new string and option settings but doesn't initiate a search. This is useful for setting values for a replace operation without executing the first Find.

Enter Selection

This command sets the search string to the current selection, clearing Grep and Multi-File Search. You can then use **Find Again** to begin searching, or **Find...** to set search options.

Find Again

This command searches for the next occurrence of a previously specified string.

Replace

This command replaces the current selection with a replacement string. If you haven't provided a replacement string, this command will clear the string that has been found (that is, it will replace it with nothing). Usually, you use this command after finding a string.

Replace & Find Again

This command replaces the current selection with the replacement string, then finds the next instance of the search string, but does not replace it. Use this command to step through a series of replacements. After each replacement, you will see the next instance of the search string, so you can decide whether you want to replace it. If you want to replace the string, use the **Replace** or **Replace & Find Again** commands. If not, use the **Find Again** command to find the next occurrence of the string.

If you haven't provided a replacement string, this command clears the string that has been found (that is, it will replace it with nothing).

Replace All

This command replaces every instance of the search string. If the Wrap Around option is on, it replaces every instance in the file. If the Wrap Around option is off, it replaces every instance from the current cursor position to the end of the file. Use this command when you don't want to give your approval for every replacement. If you haven't provided a replacement string, this command will clear the string that has been found (that is, it will replace it with nothing).

Find In Next File

This command lets you search for a string through more than one file.

To use this command, you must check the Multi-File Search check box in the **Find...** dialog box. When you check this box, another dialog box displays all of the text files known to THINK C. You can scroll through the list and select individual files by clicking on them to place a check mark by the name, or you can use the buttons in the dialog box to Check All, Check None, Check All .c or Check All .h. (If a file is already selected, clicking on its name will remove the check mark.)

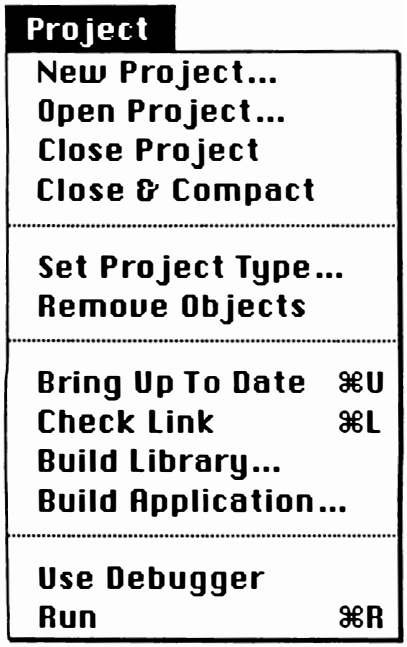
When you've checked the files you want to include in the multi-file search, click OK to return to the **Find...** dialog box.

THINK C will search for the string specified in the **Find...** dialog box through each of the files that have been checked, starting with the first one. If the search string is found in a given file, THINK C opens an edit window containing the file, and the search string is selected. At this point, you can go on and make any edits you choose. If you want to search further in the current file, you can use the **Find...**, **Find Again**, **Replace**, and

Replace All commands, which work within the current file. When you're ready to go on with the multi-file search, use the **Find in Next File** command.

Multi-file search is useful when you are writing a program, and decide to modify a function that is used in multiple files. You can open each of the files containing the search string, so you can switch back and forth between the various edit windows as necessary. (This feature is also helpful when you are tracking down link errors due to undefined or multiply defined symbols.)

The Project Menu



The commands in the **Project** menu work with the current project. You can open and create projects, set the project type, make sure all the files in the project are compiled and loaded. This menu also contains the commands you'll use when you're ready to build a file containing your application, desk accessory, device driver, or code resource.

New Project...

This command creates a new project and opens an empty project window. Only one project can be open at a time.

Open Project...

This command opens an existing project.

Close Project

This command closes the current project and then lets you open an existing project or create a new project. If you try to close a project with open files which have been

changed but not saved, a dialog box will ask you if you want to save them. To disable this dialog, turn the Confirm Saves option off in the Preferences section of the **Options...** menu. When this option is off, changed files will automatically be saved when you close.

Close & Compact

This command is the same as **Close Project**, but it makes the project document as small as possible without removing any object code. Use this command before you back up your project, or when you plan to use a project as a library (see Chapter 13) or when you plan to transmit the project through a modem (See also the **Remove Objects** command).

Set Project Type...

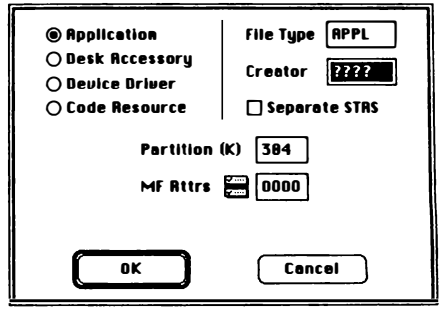
This command lets you set the project type. The default project type is Application, but you can change it to Desk Accessory, Device Driver, or Code Resource. All project types let you specify the File Type and Creator of the file created by one of the **Build...** commands. To learn the details of each project type, see Chapter 7.

Set the project type before compiling any of your sources. THINK C will need to throw away any existing object code if you switch the project type once there is compiled code in the project. If the project already contains files, THINK C will ask if you're sure you want to change the project type.

The **Set Project Type...** dialog box lets you specify different attributes for each type.

APPLICATION

The Application dialog lets you specify how string literals and floating point constants are stored and lets you set some of the fields of the `SIZE` resource MultiFinder uses.



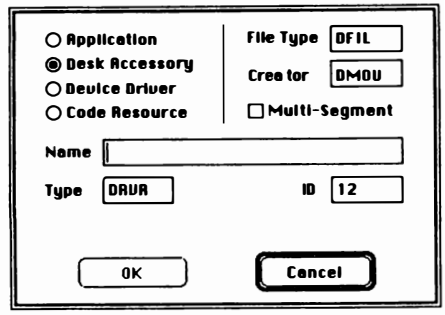
Separate STRS — When this options is set, string literals and floating point constants are placed in their own STRS component. Otherwise, strings and floating point constants are part of the DATA component.

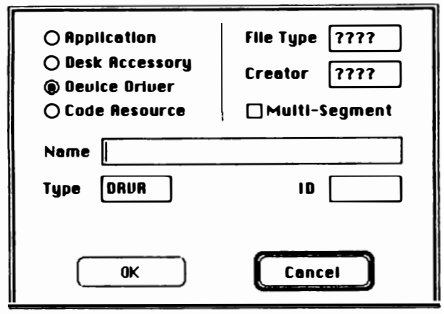
Partition — The value in this field is the amount of memory MultiFinder allocates for your application. The default is 384K, which is more than enough for most moderate size applications.

MF Attrs The pop up menu lets you set the bits that tell MultiFinder how compatible your application is. You can use the pop up menu or type a hex value into the field.

DESK ACCESSORY
DEVICE DRIVER

The Desk Accessory and Device Driver dialogs are similar. For desk accessories the File Type and Creator fields are filled in for you so the resulting file will be a Font/DA Mover file. There are other differences between Desk Accessories and Device Drivers which have to do with the header fields. See Chapter 7 for details.





Multi-Segment When this option is on, your desk accessory or device driver can have up to 30 segments.

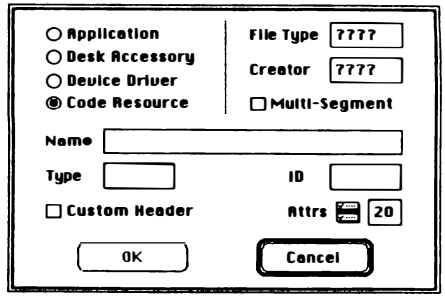
Name This field is the name of your desk accessory or device driver. By convention, desk accessory names begin with a null byte. THINK C takes care of this for you. Device drivers begin with a period. If you don't provide one, THINK C will add one for you.

Type Desk accessories and device drivers are resources of type DRVR. You can change the type if you have some reason for doing so.

ID The ID number of the DRVR resource. Desk accessories default to 12. The Font/DA Mover will renumber your desk accessory (and its owned resources) if there is a conflict.

CODE RESOURCE

The Code Resource dialog lets you specify whether your code resource will begin with a standard header, its name, type, and ID, and its resource attributes.



Multi-Segment When this option is on, your code resource can have up to 30 segments.

Name The name of your code resource. For most code resources, the name is optional.

Type The resource type of your code resource.

ID The ID number of your code resource.

Custom Header When this option is off, THINK C uses a standard header for your code resource. The header places the address of your resource in register `A0` and branches to your `main()` function. If you check this option, your code resource will begin with the first function in the file in which `main()` is defined.

Attrs The resource attributes (locked, purgeable, system, etc) of your code resource. You can select the attributes from the pop up menu, or you can set them by typing a hex number into the field.

Remove Objects

This command removes all the object code from a project. It reverses the effects of all previous compilations and loading of libraries. The project document is "dehydrated"; it can be "reconstituted" by recompiling all source files and reloading all libraries.

Use **Remove Objects** when you need to make the project document as small as possible, e.g., for archiving or for transmitting to someone else.

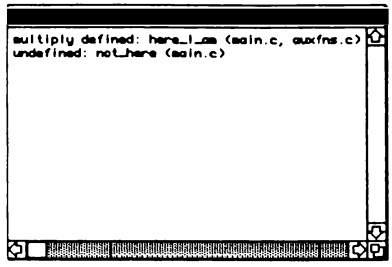
Bring Up To Date

This command compiles and source files that need to be compiled and loads any libraries that haven't been loaded.

Check Link

This command checks for all the same error conditions as **Run** or **Build Application...** would, but without running the project or building an application. If any files need to be made, you will be asked whether you want to bring the project up to date, even if you have set **Confirm Auto-Make** off.

The **Check Link** command displays errors in the **Link Errors** window. If there are multiply defined or undefined symbols, the names of the files containing the symbols appear in parentheses.



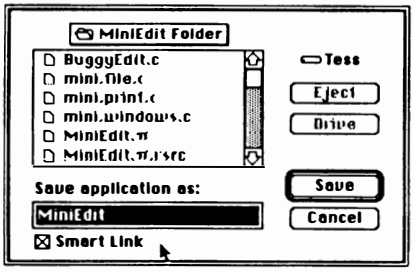
When the **Link Errors** window comes up as the result of an attempt to **Run** or **Build...**, this additional information is not displayed. Since some disk activity is required to compute the information, it is only displayed when you specifically request **Check Link**.

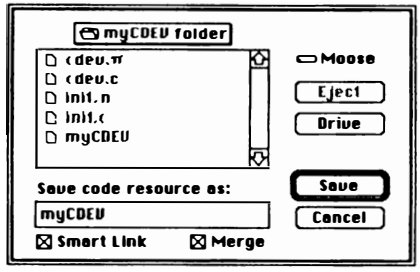
Build Library...

This command saves the current project as a single binary file that can be added as a library to other project documents. A dialog box prompts you for the name of the library file. The convention is `name.Lib`; however, a library may have any valid file name. Note that you can include a project in another project without first saving it explicitly as a library. See Chapter 13 for details.

Build Application...
Build Desk Accessory...
Build Device Driver...
Build Code Resource...

This command saves the current project as an application, desk accessory, device driver, or code resource. A dialog box lets you name the resulting file (the dialog on the left is for applications, desk accessories, and drivers; the one on the right is for code resources):





If the Smart Link option is checked, THINK C uses the smallest number of code components of the source files or libraries to create the resulting file. It takes a little longer to put the application or resource together, but the resulting file will be as small as possible. Uncheck this option if you're building frequently for testing.

If the Merge option is checked in the **Build Code Resource...** dialog, you can place your code resource into an already-existing file. Just type in the name of the file.

If there is a file with the same name of the project that ends in `.rsrc` in the same folder as the project, the resources in the `.rsrc` file will be merged into the resulting file. However, code resources built with the Merge option will not use the `.rsrc` file.

Use Debugger

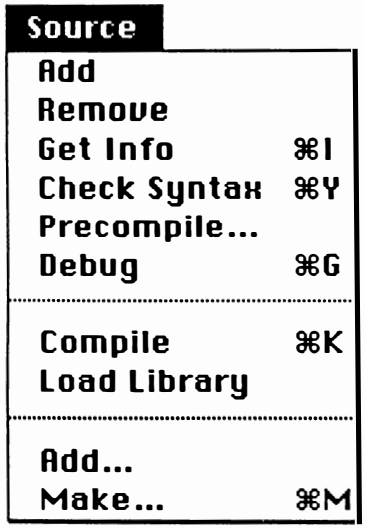
This command turns the source debugger on and off. When the source debugger is on, you'll see a "bug" column in the project window, and when you run the project the debuggers windows appear on the screen. Selecting this command is the same as clicking on the Use Debugger check box of the Source Debugger section of the **Options...** dialog box. See Chapter 11 to learn how to use the source level debugger.

Run

This command will run the program contained in the project. If the project is not up to date, a dialog box will ask if you want to bring it up to date. If the **Confirm Auto-Make** option in the **Options...** dialog box is not checked, then the project will automatically be brought up to date. This lets you edit a source file and run the changed program, without ever explicitly recompiling or relinking the program.

If the project type is Desk Accessory, the **Run** command uses an auxiliary program, DAShell, to run your desk accessory. THINK C needs to build you desk accessory and save it in a file first, then THINK C launches DAShell. Your desk accessory will be in the Apple menu.

The Source Menu



The commands in the **Source** menu let you add and remove source files to your project. This menu also contains commands to create precompiled headers, to compile source files, to load libraries, and to control the auto-make facility yourself.

Add

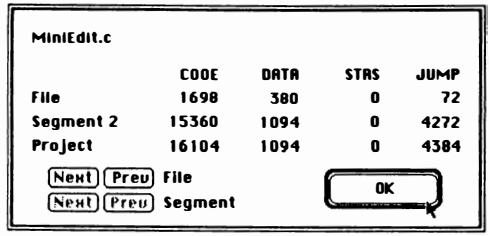
This command adds the file in the frontmost edit window to the project to the project window. The file must end in `.c`.

Remove

This command removes a selected source file or library from the project.

Get Info...

The Get Info... command display a dialog box that contains the sizes of each project component. The displays initially contains information for the currently selected file.



The dialog shows the size of the CODE, DATA, STRS, and JUMP components for the selected file. If the file hasn't been compiled, these values will be zero. The dialog also shows the same information for the segment and the entire project. The Next and Prev buttons let you examine the other files and segments in the project.

Check Syntax

This command lets you compile a file in order to check its syntax. This command compiles the front window but does not add the file to the project window or post the results of the compilation to the project document. You can check the syntax of the contents of the edit window, even an untitled window. **Compile**, by contrast, works only on files that end in .c.

Precompile...

This command creates a precompiled header from the contents of the frontmost edit window. Precompiled headers may not have any code or data definitions. You can include #include files (even other precompiled headers) in precompiled headers.

Debug

When you're using the source level debugger, this command sends the frontmost edit window (or the selected file in the project window) to the Source window of the debugger.

Compile

This command will compile either the contents of the active edit window, or the currently selected file in the project window. The results of the compilation will be posted to the project document.

Only files that end in .c can be compiled. To check the syntax of a source file without adding it to the project document, use the **Check Syntax** command instead. The **Compile** command is dimmed when a library file is selected in the project window.

Load Library
Load Project

These commands are active when the selected file in a project window is a library or a project. (You can use projects as libraries. See Chapter 13 for details.) When you select this command, THINK C loads the code for the library into the project. To add a library to a project for the first time, use the **Add...** command.

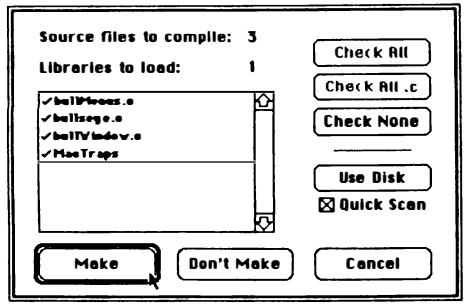
Add...

This command lets you add existing source files and libraries to a project. This command displays a standard file box and lets you select a source file, library, or project to be added to the project. It keeps asking for more until you press the Cancel button. Only files which can be added (i.e., source files that end in .c, libraries, or other projects) appear in the scroll box.

Make...

When you select this command, a dialog box appears showing all the files in the project inside a scroll box. The files that THINK C thinks need to be recompiled (or in the case of libraries, reloaded) are checked. You can alter the list if you like by using the cursor to check or uncheck files, or by clicking the Check All, Check All .c or Check None buttons. Your changes will not be remembered if you click the Cancel button.

When you press the Make button, THINK C will bring the project up to date. It will recompile files that need to be recompiled and load libraries that need to be reloaded. If you press the the Don't Make button, the project will be updated next time you use any of the commands that the update the project: **Bring Up To Date**, **Run**, **Check Link**, or **Make...**.



If you click the **Use Disk** button and the Quick Scan check box is checked, THINK C checks the date/time-modified of all the files in the project. Normally this is not necessary because THINK C automatically tracks the changes you make as you edit. This knowledge is project-specific, though, so if you have a source file that belongs to two different projects and you change it in one, the other project won't know it's been changed unless you say Use Disk.

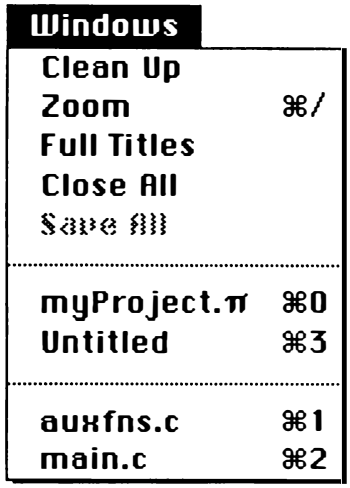
If the Quick Scan check box is not checked, THINK C does a more extensive check. If a file can't be found, THINK C searches the tree the file was originally in to find the file. (The Use Disk feature can't help you detect when you've moved files from one tree to another. See Chapter 9 for details.)

Use Disk displays its progress if the Quick Scan option is off. You can abort it by typing Command-. (command-period), although the next **Use Disk** will start again at the beginning.

If a source file or library is not found by Use Disk, its status (i.e. whether it is checked in the **Make...** box) is unchanged. However, if a source file is found but one of the files it includes is not found, the source file is marked as needing to be compiled (i.e., it is checked).

Clicking Cancel does not undo the effect of Use Disk. Unlike the other buttons, which simply add (or remove) a check mark to those specified by Auto-Make, Use Disk actually updates the date/time record associated with each file that is in the project. You can, of course, manually check or uncheck files after telling THINK C to Use Disk.

Windows Menu



The **Windows** menu has three sections, separated by dotted lines. The first section has five commands: **Clean Up**, **Zoom**, **Full Titles**, **Close All**, and **Save All**. The second section has an entry for the project window and one for each Untitled window. The third section has an entry for each file open in an edit window, in alphabetical order.

Clean Up

Restacks the windows as though they were freshly opened. The rearmost window is assigned the first slot, as though it was the first window opened, the next-rearmost window is assigned the second slot, etc. The front-to-back order of the windows is not changed.

Zoom

Resizes the frontmost window to occupy the full screen. If the window already at full screen, it is restored to its previous position and size. This is the same as clicking the window's zoom box in the upper right corner of the window.

Full Titles

This is a checkable item, initially unchecked. When checked, the title of each edit window indicates the volume name and directory name as well as the file name.

Close All

This command closes all the edit windows. If the Confirm Saves option is checked, THINK C asks whether you want to save each modified window, otherwise windows are automatically saved. Holding down the Command or Option key as *you click in* the close box of an edit window is the same as **Close All**.

Save All

Saves all the modified windows. No confirmation is requested.

Project window

The project window is brought to the front. (The menu item will be the name of the project, not the literal words "Project window".)

Titled and Untitled edit windows

Brings the selected window to the front. The number reflects the "slot number", i.e., the initial position of the window. (The first created window occupies slot #1, and slots #6-10 occupy the same screen positions as slots #1-5, etc. Slots vacated by closed windows are reused at the next opportunity.) The number of windows is limited only by available memory, but only windows in the first nine slots have a Command-key equivalent. A diamond (◊) appears next to windows which have been modified.

Debugger Menus
56

Introduction

This chapter describes each of the source level debugger's menu commands. It is organized by menu, from left to right along the menu bar. Within each menu, commands are described in the order in which they appear in the menu.

Some of the debugger's commands operate on the selected statement. To select a statement, just click on its line in the Source window. If there isn't a selected statement, the commands operate on the current statement.

The Menu

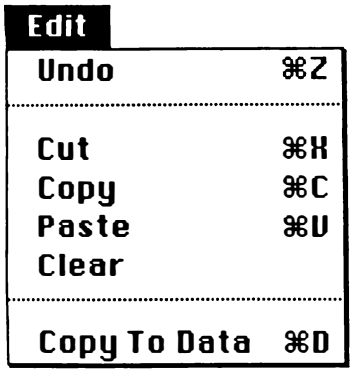
Shortcuts...

This command displays a series of dialog boxes that describe some shortcuts that make working with the debugger faster.

The File Menu

Unlike virtually every Macintosh application, the source level debugger does not have a File menu. This menu is always dimmed. The reason this non-menu still occupies a slot is to let you know at a glance that one of the source level debugger's windows is the active window.

The Edit Menu



The **Edit** menu lets you use the standard editing commands on expressions in the Data window.

Undo

This command undoes any edits you make to an expression in the Data window. Choosing this command is the same as clicking on the deselect button and then selecting the same expression again.

Cut

This command deletes the selected text and copies it to the Clipboard. You can't cut text out of the Source window.

Copy

This command copies the selected text to the Clipboard.

Paste

This command pastes the text in the Clipboard into the current window. You can't paste into the Source window.

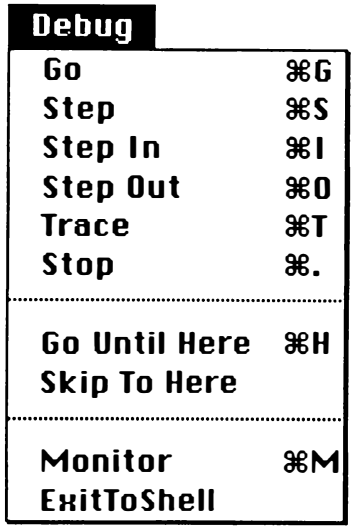
Clear

This command removes the selected expression from the Data window.

Copy To Data

This command is active only when the Source window is the frontmost window. It copies a selected expression from the Source window and pastes it into the Data window, where the expression is compiled. It's up to you to make sure that the text selected in the Source window is a valid expression. Copy To Data leaves the expression selected, so you can use some of the formatting commands right away.

The Debug Menu



Use the **Debug** menu to control the execution of your program. The first six commands in this menu have equivalent buttons in the Source window status panel.

Go

The **Go** command starts your program if it was stopped. Your program will run until you stop it (with the Stop button, for example) or until it's about to execute a line with a breakpoint or until it hits an exception (dividing by zero, for instance). If your application is already running, the Go command brings it to the foreground.

Step

The **Step** command goes on to the next statement marker in the current function. If you're at the end of a function, **Step** returns to the calling function. Use **Step** when you want to follow the execution within a function without falling into other functions. (Technically speaking, Step skips over JSRs.)

Step In

The **Step In** command executes your program until the current statement arrow falls into a function. **Step In** is useful when you want to skip over a set of assignments or Toolbox traps, to fall into the next function call. If **Step In** reaches the last statement of the current function without falling into another function, it will stop immediately after the current function returns.

Step Out

The **Step Out** command executes your program until the current statement arrow falls out of the current function. This operation can be slow if there's a lot left to do, but it's a sure way

of leaving the current routine. A faster way of leaving the current routine is to use the **Go Until Here** command or to set a temporary breakpoint at the last diamond in the function.

Trace

The **Trace** command executes the current statement. In most cases, the statement indicator will go on to the next statement marker, even if the next statement marker is in another function. The only time it won't is when the program counter steps into some code that the debugger doesn't have the source text for. This usually happens when you step into a trap that's not generated in line. So, for a brief period, the current statement arrow isn't really anywhere in your program, but somewhere in `MacTraps` instead. Though you can't see the current statement arrow, the current function indicator at the lower left tells you which file it's in.

Stop

The Stop command stops execution of your program. The Stop command works when any debugger window is active, and the Stop button only works when the source window is the active window. When you press the Stop button you'll usually be coming out of your call to `GetNextEvent()` or `WaitNextEvent()`.

If your program is not in its event loop, you might not be able to make the debugger the frontmost application. In this case, Command-Shift-Period is the **panic button**. Use Command-Shift-Period to stop execution when one of your application's windows is frontmost or when you think its stuck in a loop. (Command-Shift-Period won't work if you're stuck in an infinite loop in ROM, though.)

Go Until Here

The **Go Until Here** starts execution and stops at the selected line. This command is exactly the same as setting a temporary breakpoint (see "Setting Breakpoints" in Chapter 11) at the selected line. Use this command when you want to move through a block of code quickly.

Skip To Here

The **Skip To Here** command changes the program counter to the selected line without executing any intervening code.Use it when you want to skip over code you know to be buggy but not crucial to the rest of the program's operation.

> **Note:** This command is potentially dangerous. Make sure the code you're skipping to doesn't depend on anything the skipped code does. For instance, it is a very bad idea to skip over initialization routines.

Monitor

The **Monitor** command drops you into a low level debugger. All your registers (including status registers) and low memory globals will be correct. The PC (program counter) will be somewhere in the source debugger, not in your program. You can still get the value of your PC.

If you're using TMON, the PC is in TMON's V register. If you selected an expression in the *Data* window, its value is in TMON's N register.

If you're using Macsbug, your PC is one long word before the current PC. To look at the instructions in your program, type:

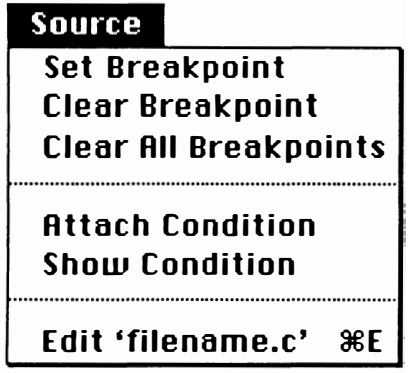
```
DM PC-4
IL @.
```

> **Note:** If you don't have a low level debugger installed, don't use the **Monitor** command.

ExitToShell

This command aborts the source level debugger. You should use your application's **Quit** command to quit the debugger. Use **ExitToShell** only if you can't use your application's **Quit** command.

The Source Menu



The **Source** menu contains commands for working with the Source window.

Set Breakpoint

Sets a breakpoint at the selected statement.

Clear Breakpoint

Clears the breakpoint at the selected statement

Clear All Breakpoints

Clears all the breakpoints in the project.

Attach Condition

Use the **Attach Condition** command to attach an expression in the Data window to a breakpoint to create a conditional breakpoint. To set a conditional breakpoint:

- Set a breakpoint by clicking on the statement marker diamond
- Click on the line to select it
- Click on an expression in the Data window
- Choose **Attach Condition** from the **Source** menu.

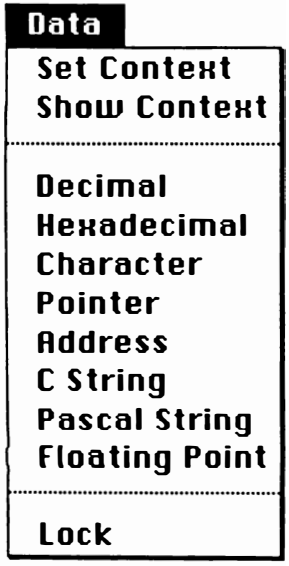
Show Condition

Use the Show Condition command to display the condition attached to a conditional breakpoint. If the selected statement has a conditional breakpoint (a gray diamond), the attached expression in the Data window will be highlighted.

Edit 'filename.c'

Brings THINK C to the foreground and opens an edit window for the file in the Source widow. This is the inverse of the **Debug** command in THINK C's **Source** menu.

The Data Menu



The commands in the Data menu operate on expressions in the Data window. You can set and show the context of expressions, change their display format, and lock expressions to keep them from being reevaluated.

Set Context

This command makes the selected statement in the Source window the context of the selected expression in the Data window.

Show Context

This command highlights the statement that is the context of the expression selected in the Data window.

Decimal
Hexadecimal
Character
Pointer
Address
C string
Pascal string
Floating Point

These commands control how expressions appear in the Data window. The default format depends on the data type of the expression. The type of the expression also determines what other formats you can use.

The default formats are shown in italics.

Type	Formats Available
integers	*decimal*, hex, char
unsigned	*hex*, decimal, char
pointers	*pointer*, address, hex, C string, Pascal string
arrays	*address*, C string, Pascal string
structs	*address*
unions	*address*
functions	*address*
floats	*floating point*

This is what the display formats look like:

Format	Example
decimal	`4523345, -23576`
hex	`0xA09E1487`
char	`'c', 'TEXT'`
C string	`"abcdef\nghi\33"`
Pascal string	`"\pabcdef\nghi\33"`
pointer	`0x7A7000`
address	`[] 0x09FE44, struct 0x08FC14`
floating point	`1961.0102`

The C string and Pascal string formats display non-printing characters in backslash form. Whenever it can, the debugger uses the built-in escape characters (`\n`, `\r`, `\b`); otherwise it uses `\nnn`, where nnn is an octal value.

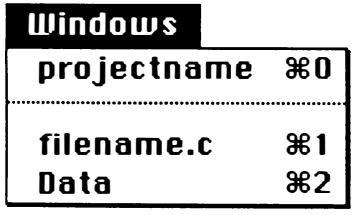
Of course, you can use type casting to use formats that aren't normally available. For example, if you really wanted to see an integer, `i`, as a C string, you would type this expression: `(char * ) i`.

Lock

The debugger reevaluates all the expressions in the Data window every time execution stops. To keep an expression from being reevaluated, select it, then choose **Lock**. When an expression is locked, a small lock icon appears next to it.

To unlock an expression, select the expression, and choose the **Lock** command again.

The Windows Menu



The commands in the **Windows** menu work on the source debugger's windows.

projectname

The real name of this command is the name of your project. Choose this command to bring THINK C to the foreground and make the project window the active window.

filename.c

The real name of this command is the name of the file displayed in the Source window. When you choose this command, the Source window becomes the active window.

Data

This command makes the main Data window the active window.

Language Reference 57

Introduction

This section describes the C language as implemented in THINK C. You should read this chapter in conjunction with Appendix A of Kernighan and Ritchie's *The C Programming Language, Second Edition*. The sections are named and numbered exactly as they are in *The C Programming Language, Second Edition*. Only the cases where THINK C differs are listed here.

> **Note:** *The C Programming Language, Second Edition* is based on a draft of the ANSI standard. The reason this manual uses *The C Programming Language, Second Edition* rather than the standard for this chapter is that it is more widely available.

2.3 Identifiers

In THINK C, there is no limit to the number of characters which are significant in an identifier. In THINK C, no additional restrictions apply to external identifiers.

2.4 Keywords

The following identifiers are reserved in THINK C:

```
asm         double      int         struct
auto        else        long        switch
break       enum        pascal      typedef
case        extern      register    union
char        float       return      unsigned
continue    for         short       void
default     goto        sizeof      while
do          if          static
```

The keywords `asm` and `pascal` are reserved in THINK C but not in ANSI C.

The keywords `const`, `signed`, and `volatile` are not reserved in THINK C.

2.5.1 Integer constants

THINK C does not support the `U` or `u` suffix for integer constants. THINK C uses different rules to determine the type of integer constants. Decimal constants without suffixes are of

type `int`, or if it won't fit in an `int`, `long int`. Octal and hexadecimal constants are always `unsigned int` or `unsigned long int`.

2.5.2 Character constants

THINK C supports all of the character constants and hex character constants defined in *The C Programming Language, Second Edition.*

A character literal is always of type `int`. Single characters with the high bit set will have negative values. For instance, `'\377'` is –1, not 255.

THINK C does not support the `L` prefix for extended character sets.

Multi-character character constants are allowed. The type of such a constant is `int` if the value is in range, `long int` otherwise. The characters are assigned left-to-right and right-justified. More than 4 characters are not allowed.

2.5.3 Floating constants

Floating constants have type `double`.

THINK C does not support the `F` suffix for floating constants.

2.5.4 Enumeration constants

All of the enumerations constants in a enumeration have the same type. If the values of all of the enumeration constants fall in the range –128 to 127, the type of the enumeration constants is `char`, other wise their type is `int`.

2.6 String literals

A string beginning with `"\p"` or `"\P"` is a Pascal string. It is not terminated with a null byte; instead the first byte is a length byte indicating the number of characters following. Be careful; the length byte may appear negative if it exceeds 127, and it will not be meaningful at all if the string is longer than 255 characters.

Pascal strings are required when calling certain Macintosh routines; however, if you like, they can be used in other cases as well.

THINK C does not support the `L` prefix for extended character sets.

All string constants are aligned on word boundaries.

4.2 Basic types

Variables of type `char` are always signed in THINK C. Data types have the following hardware characteristics in THINK C:

type	size
`char`	`8 bits`
`short int (short)`	`16 bits`
`int`	`16 bits`
`long int (long)`	`32 bits`
`float`	`32 bits`
`short double`	`64 bits`
`double`	`80 bits (96 bits if 68881 option is on)`

Floating-point is IEEE standard, courtesy of Apple's Standard Apple Numeric Environment (SANE) numerics package. The range of `double` values is $\pm 10^{\pm 4932}$. `float` corresponds to SANE `SINGLE`, `short double` corresponds to SANE `DOUBLE`, and `double` corresponds to SANE `EXTENDED`.

4.4 Type qualifiers

THINK C does not support the `const` and `volatile` type qualifiers.

6.1 Integral promotion

THINK C does not use the ANSI rules for integral promotion. THINK C preserves the signedness of the values in an expression.

6.5 Arithmetic conversion

In THINK C all floating point expressions are of type `double`. Floating point expressions are done in 80-bit precision (96-bit if the MC68881 option is on).

In expressions involving a small unsigned integer with a longer signed integer, the result is always unsigned.

In expressions all operands of certain types are converted to a larger type as follows:

type	converted to
`char`	`int`
`unsigned char`	`unsigned int`
`float`	`double`
`short double`	`double`

Then, if both operands have the same type (or if there is only one operand), that is the type of the result. Otherwise, both operands are converted to a common type, and that is the type of the result. The common type is whichever of the two types appears *first* in the following list:

```
double
unsigned long int
long int
unsigned int
int
```

7.4.8 Sizeof operator

In THINK C, the type that `sizeof` returns is `int`, not `size_t`.

7.7 Additive operators

In THINK C, the type of the difference between two pointers is type `long int`.

7.8 Shift operators

In THINK C, a right shift of a signed quantity preserves the sign. In other words, for signed quantities, THINK C performs arithmetic shifts; for unsigned quantities, THINK C performs logical shifts.

8.1 Storage class specifiers

THINK C implements the additional storage class `pascal`. An identifier declared `pascal` must have type "function returning ...". A `pascal` function is called using Pascal calling conventions. If the `pascal` keyword appears in a function definition, the function will expect to be called using Pascal calling conventions. The `pascal` keyword may be used with `extern` and `static`.

8.2 Type specifiers

THINK C does not support the `signed` type specifier.

8.3 Structure and union declarations

Members of all types other than `char`, `unsigned char`, and singly and multiply dimensioned arrays of `char` or `unsigned char` are aligned on even addressing boundaries. Similarly, `structs` and `unions` are padded to an even size.

It is legal to specify an array without size as the last member in a `struct`. The array does not contribute to the size of the structure. For example:

```
struct {
        unsigned int    count;
        char    data[];
} CountData; /*  size of CountData is 2  */
```

Bit-fields may be declared to be of any integral type. The size of the declared type determines the word size for that bit-field; thus a word may be 8, 16, or 32 bits in length. Keep this definition of "word" in mind throughout the following discussion.

A sequence of bit-fields with the same word size are packed into a word, but a bit-field is placed in the next word if it would otherwise straddle a word boundary. No bit-field may be wider than a word. Fields are assigned beginning with the high-order bit of a word. An unnamed field with a width of 0 "closes out" the current word. A bit-field with a different word size from the preceding bit-field causes this to happen automatically (just as a non-bit-field member does).

The high-order bit of a bit-field is *not* treated as a sign bit, even if the declared type of the bit-field is signed. Think of it this way: the "real" type of the bit-field is "`unsigned` *n* bits", which gets converted to the declared type whenever the bit-field appears in an expression.

8.4 Enumerations

The type of an enumeration is `char` if all the enumeration constants of the enumeration fall in the range –128 to 127. Otherwise the type of an enumeration is `int`.

8.6.3 Function declarators

Prototype arguments which are not associated with a named function are permitted, but not honored. For example:

```
int (*fn)(int n);                       /* prototype ignored */
typedef Boolean Func(long);             /* prototype ignored */
void f(int x, int (*p)(double));        /* prototype for f is honored, */
                                        /* prototype for p is ignored  */
```

10.1 Function definitions

In THINK C `floats` are always passed as `double`.

10.2 External declarations

THINK C lets you declare variables `extern` as many times as you wish. Any declaration not declared `extern` becomes the definition of the variable.

THINK C does not allow an initializer when a variable is declared `extern`; nor does it allow more than one declaration that is not `extern`. In other words, THINK C *does not* support the notion of tentative definition.

11.1 Lexical scope

THINK C uses file scope rules for variables declared `extern` within a function.

12.1 Trigraph sequences

THINK C does not implement the trigraph sequences defined by ANSI.

12.1 Line splicing

THINK C implements line splicing only for string literals and macros. For macros, the newline is preserved instead of being folded into the stream.

THINK C does not support \ *newline* in any other context.

12.4 File inclusion

THINK C uses a convention to prevent #include files from being included more than once in the same file. If an #include file named `filename.h` defines a symbol `_H_filename`, the #include file is included only once.

12.5 Conditional compilation

THINK C allows the use of `sizeof` in `#if` and `#elif` directives.

12.8 Pragmas

THINK C does not support any pragmas. Any `#pragma` directives are ignored.

12.10 Predefined names

The identifier `__STDC__` is not defined in THINK C. The symbol `THINK_C` is defined as 1 in THINK C.

THINK C

PART SIX

Appendices

A The Profiler
B Troubleshooting
C Error Messages
D RMaker Reference

The Profiler
A

Introduction

This chapter shows you how to use the THINK C code profiler. The profiler is a tool that records how much time your functions take to execute. The profiler uses the `ANSI` library. The profiler source code is part of your THINK C package, so you can modify to suit your task.

Topics covered in this chapter:

- Using the profiler
- Modifying the profiler

Using the Profiler

To use the profiler, check the Profile check box in the Code Generation section of the **Options...** dialog. When this option is on, THINK C inserts calls to profile routines at the beginning and end of your functions.

> **Note:** The Profiler always generates a stack frame, even for functions with no parameters and no local storage.

Your project must contain the `profile` library as well as the `ANSI` library. Finally, put a call the `InitProfile()` function in your program to start the profiler. This takes two arguments. The first `nsyms` is the maximum number of functions that the profiler will track. The second `depth` is the maximum call depth. A good default for these arguments is 200.

The profiler logs the amount of time spent in each function, and prints the results to `stdout` on exit. The display reports on:

- minimum time spent in routine
- maximum time spent in routine
- average time spent in routine
- percentage of profiling period spent in the routine
  (The profiling period is the accumulated time spent in routines that were compiled with the profile option on.)
- number of times the routine was called

The profiler uses the units of the VIA 1 timer. Each unit is 1.2766 µsec. You can change the units to ticks (60ths of a second) by editing the profiler code. See "Modifying the Profiler" below.

Turning the profiler on and off

The profiler prints its report to `stdout` when your program exits. (It uses the `onexit()` function.) To see the statistics any time before your program ends, just call the function `DumpProfile()`.

You can control whether the profiler is accumulating time statistics by setting the value of the `_profile` variable. If this variable is zero, the profiler stops recording time information.

The profiler can print an indented call-tree of your functions calls as your program runs. Set the `_trace` variable to non-zero to have the profiler print every time a function is entered.

Modifying the Profiler

You can modify the profiler code to change its behavior. The profiler sources are in the `sources` folder of the `C Library` folder in disk `THINK C 2`. The profiler consists of two files: `profilehooks.c` and `profile.c`.The `profilehooks.c` file contains the two assembly language routines that are called at the beginning and end of each function. These routines call the higher level profiler routines in profile.c which do the statistics collection.

The symbol `VIATIMER` in `profile.c` controls whether to use the VIA timer or the tick counter. This symbol is initially defined.

> **Note:** When you rebuild the `profile` library, make sure that the Profile option is not selected. Otherwise you will be in an infinite loop at run time.

Summary

`InitProfile()`	(`profiler.c`) Start profiling. Takes two arguments: `nsyms`, the maximum number of functions to track, and `depth` the maximum call depth.
`DumpProfile()`	(`profiler.c`) Write the statistics collected so far to `stdout`. You can call this function any time to print the current statistics. It is called automatically on exit.
`_profile_()`	(`profilehooks.c`) Assembly language routine called at the beginning of each function when the Profile option is on. When this routine is called, the stack contains the address of a Pascal string that is the name of the function. `_profile_()` changes the return address of the routine to be `_profile_exit()`.

	that is the name of the function. `_profile_()` changes the return address of the routine to be `_profile_exit()`.
`__profile()`	(`profile.c`) C function that collects timing statistics.
`_profile_exit()`	(`profilehooks.c`) Assembly language routine called at the end of a function. Code to execute this routine is not generated, instead, `_profile()` munges the stack so the original routine "returns" to this one.
`__profile_exit()`	(`profile.c`) C function that computes timing statistics.
`_profile`	(`profile.c`) Variable, if non-zero, enables profiling.
`_trace`	(`profile.c`) Variable, if non-zero, enables call tree generation.

Troubleshooting B

Introduction

This chapter contains some frequently asked questions, and a set of tips and suggestions for avoiding errors and improving code.

Topics covered in this chapter

- Getting help
- Some common problems

Getting Help

If you run into a problem, look in this chapter to see if it's covered here. If it's not, look in the section of the manual that deals with the part of THINK C that's giving you trouble. For technical support in North America call (617) 275-1710. In the United Kingdom, call Symantec (UK) Ltd at (0628) 776343. Outside North America or the UK please contatct the exclusive Symantec distributor in your local area.

Some Common Problems

You might run into some common problems when you use THINK C. Most of these problems are easy to correct.

Undefined symbols

Undefined symbol errors show up in the Link Errors window when THINK C can't link your project during a **Run**, **Build...**, **Bring Up To Date**, or **Make...** command. Typically, you'll see a message like `undefined: printf`.

- You have not added a necessary library. If the missing symbol is a Macintosh Toolbox function, you probably forgot to add the `MacTraps` library to your project. The `MacTraps` library must be in nearly all projects, since it contains the QuickDraw globals and glue code that accesses the register-based Macintosh Toolbox functions, as well as the `[Not in ROM]` routines.

- If you're using any C library routines, you may have forgotten to add the library that contains that routine. Check the documentation for the routine in the *Standard C Libraries Reference*. The name of the library you need is given with the description of each routine. Use the **Add...** command to add the library to your project.

- You may have made a spelling or capitalization error. C is a *case-sensitive* language, which means that `printf` is not the same as `Printf`. Check to see that the undefined function name is spelled and capitalized correctly.

- You have redefined a Macintosh or C library function as "`extern`". For example:

```
extern void SetRect();
```

  Such re-definitions are unnecessary, since Toolbox function definitions are already built into THINK C. If one appears, the linker will try to resolve the reference with a user-defined function. This is fine if what you want to do is *replace* a standard function, but will cause a linker error if no replacement function is provided. (If you want to supply return type or prototype information for Toolbox calls, see Chapter 10.)

Code segment too large

If the Link Error window reports `Code segment too big`, one or more of your segments contains more than 32K of object code.

- Move files out of the too-big segment until it contains less than 32K of code. See the "Segmentation" section of Chapter 7 to learn how to move files between segments. Use the **Get Info...** command in the **Source** menu to see how large your segments are.

Data segment too big
Stack frame too big

The static and global data area of THINK C applications is limited to 32K *total* per project. The limits for desk accessories, device drivers, and code resources are different. See Chapter 7 for details.

- If you want to be able to access global or local memory larger than the 32K limit, allocate the memory dynamically as a pointer or handle. Because the C language blurs the distinction between arrays and pointers, you can then use the pointer with an array notation to access elements in dynamic memory. (Refer to a good C language manual, such as Kernighan and Ritchie's *The C Programming Language* for more details.) For example:

```
#define SIZE     100000
#define SIZE2D       100

int BigBadArray[SIZE];                /* a too big array */
int *LooksLikeAnArray;
int BigBad2DimArray[SIZE2D][SIZE];    /*too big 2D array*/
int *LooksLikeA2DArray[SIZE2D];       /*array of pointers*/
```

```
proc()
{
 register long i;

 /* allocate "array"*/
 LooksLikeAnArray = (int *)NewPtr(sizeof(int)*SIZE);

 /* You can index it like an array */
 LooksLikeAnArray[60000] = 5;

 /*  allocate 2D "array"*/
 for (i = 0; i < SIZE2D; i++)
 LooksLikeA2DArray[i] = (int *)NewPtr(sizeof(int)*SIZE);

 /* You can index this like an array, too */
 LooksLikeA2DArray[10][20] = 7;
}
```

Can't find an #include file

When THINK C reports that it can't find an #include file, the file is probably not in the right tree. Your project's #include files should be in your project tree, and standard #include files should be in the THINK C Tree. To learn about the two trees, see Chapter 9.

Odd address error (System error ID=02)

An odd address error usually means that your program is accessing memory on the heap incorrectly. These errors can usually be traced to a few common causes.

- You may have referenced a null handle or pointer. If your program uses a resource file that is not found, a `GetResource()` (or `GetNewMenu()`, `GetNewWindow()`, etc.) call will return a null handle. Make sure that if you use a resource file, it's in the same folder as your project file, and that it is named *`projectname`*`.rsrc`. (See Chapter 7.)

- You may not have enough memory for a `NewHandle` or `NewPtr` call, resulting in a null return value. Be sure you check the results of every memory allocation.

- You may have not initialized a pointer variable.

- You may have a Toolbox routine with bad or inappropriate data. For example, you are likely to crash if you call `DisposHandle()` with a handle to a resource. Use `ReleaseResource()` instead.

Printf and scanf not working

It might seem that the standard C functions `printf()` and `scanf()` don't work correctly.

- A common misconception about `printf()` is that `printf()` "knows" about its arguments. If you pass a long expression to `printf()`, you must specify in the format string that you want a long expression to be printed. Example:

```
int anInt;
long aLong;
printf("long is %ld, int is %d\n", aLong, anInt);
```

In `scanf`, the same goes for floats and doubles:

```
float aFloat;
double aDouble;
printf("Input a double and a float:");
scanf("%lf %f", &aDouble, &aFloat);
```

Link error: printf undefined

Many novice users (and not-so-novice users!) get confused about the difference between "header" files and "library" files. Header files (which conventionally have names which end in `.h`) contain source statements: definitions and declarations which allow the *compiler* to make sense of source code calls to a library function. Libraries contain the actual object code for the functions themselves, which the *linker* references in building the project. Including a header file is for the compiler only; it tells the linker nothing. `#include`-ing a library in a source file is an error.

Error Messages C

This appendix is a guide to the error message THINK C generates.The error messages are arranged in alphabetical order. Error descriptions marked "Assembly" refer to errors generated by the inline assembler.

8-bit reference to *'symbol'* out of range

(Assembly) You've used a symbol that's out of range for an assembly language instruction that expects an 8-bit operand. You'll get this error if you try to do a short branch to a label that's over 127 bytes away from the branch instruction. Because the inline assembler optimizes branches to `goto`s, you'll also get this error if the *final* destination of the branch is over 127 bytes away. Example:

```
asm {
        bra.s   @foo
        ...     /* over 127 bytes */
foo:

kansas: bra.s   @cLabel
}

cLabel: goto landOfOz
        ...     /* over 127 bytes from kansas */
landOfOz: printf("We're not in kansas anymore!");
```

Keep in mind that `return`, `continue`, and `break` are treated just like `goto` in the inline assembler.

'&' (address-of) operator illegal here

The address-of operator was used on an object (such as a register variable or a bitfield) that doesn't have an address. Example:

```
function()
{
  register int i;
  int *p = &i;     /* Illegal - i is a register */
}
```

'symbol' has not been declared

The identifier *symbol* has been used before it has been declared.

'symbol' is not a formal parameter

The *symbol* was declared as if it was a formal parameter in a function definition, but it is not in the parameter list. Example:

```
function()
int left_out_of_parameter_list;  /* Illegal */
{
}
```

'symbol' was not placed in a register

(Assembly) You'll get this error if you THINK C was unable to place a variable in a register and you tried to use it in a register context in an inline assembly language section. Example:

```
function()
{
    register int a, b, c, d, e, f;

    asm {
        move    a,(f)        ; f isn't in a register
    }
}
```

'filename' is not a text file

A file name in an #include statement refers to an existing file that is not a text file. You'll also see the message when you Option double-click on a symbol to find its definition, and the symbol is defined in a library.

addressing mode requires 68020

(Assembly) You're trying to use an MC68020 addressing mode, but MC68020 inline assembly is not enabled. See Chapter 12.

"array of functions" is not allowed

An array of functions is not allowed. However an array of pointers to functions is allowed. Example:

```
int array_of_functions[]();                 /* Illegal */
int (*array_of_ptrs_to_functions[SIZE])(); /* Legal */
```

"array of void" is not allowed

Since `void` objects are not allowed, arrays of `void` objects are not allowed either. However, it is legal to have an array of pointers to `void`. This is a C idiom for an array of generic pointers. Example:

```
void abyss[SIZE];               /* Illegal */
void *generic_ptr_array[SIZE]; /* Legal */
```

bad operand

(Assembly) An assembly language operand has an operator that is not legal in assembly language. Example:

```
asm {
    move        d0, x->field    /* use OFFSET() macro, instead */
}
```

break outside of loop or switch

The `break` statement must be inside a loop (`while`, `do-while`, `for`) or a `switch`. Example:

```
function()
{
    int i;
    break;              /* Illegal */
    while(1)
        break;          /* Legal */
    do { break; }       /* Legal */
        while(1);
    switch(i){
    case 1:
        break;          /* Legal */
}
}
```

call of '*symbol*' does not match prototype

The prototype given for symbol and the call for function symbol don't match. Example:

```
extern function(int, char); /* the prototype */

another_function()
{
    function(23, "skidoo");     /* types don't match */
}
```

call of non-function

An expression in the function call position does not evaluate to a function. Example:

```
function(f_ptr)
int (*f_ptr)();   /* f_ptr is a pointer to a function */
{
  int i, j, k, result;
  /* probably left out an operator between i and (j+k) */
  result = i(j+k);         /* Illegal */
  result = f_ptr(i, j);    /* Illegal */
  result = (*f_ptr)(i,j); /* Legal */
}
```

can't define classes in this project

You'll get this message if you try to define a class in a single-segment desk accessory, device driver, or code resource. To use classes in these kinds of projects, you need to turn on the Multi-Segment option in the **Set Project Type...** dialog, even if your project has only one segment.

can't do that with multi-segment project

What you're trying to do is not legal when the project contains more than one segment. You'll see this message when you use the **Build Library...** command on a multi-segment project, when you use a multi-segment project as a library, or when you use the **Build Device Driver...**,**Build Desk Accessory...**, or **Build Code Resource...** command on a project when the Multi-Segment option is not on.

can't load STRS in this project

You can't load a library or project built with the Separate STRS option on into a project whose Separate STRS option is off. You can rebuild the library with the Separate STRS off, or you can turn the option on in the project that's using the library.

can't open #include'd file

The file to be `#include`'d cannot be opened. Its name may be misspelled, it may not exist, or it may be in the wrong tree (see Chapter 9).

can't precompile code or data

You tried to Precompile a file that contained a function or variable definition. Only declarations are allowed in precompiled headers.

case not in switch

The `case` keyword was found outside of a `switch` block.

code overflow

This message means that you have more than 32K of code or data in one file. Break your file up into smaller files.

constant required

A constant was required, but none was found. Example:

```
#define aConstant  12
function()
{
  int h;
  int x[2];  /* Legal - array dimension is a constant */
  int y[h];  /* Illegal - h is not a constant */
  enum{
     color = h, /* Illegal - enum values must be constant */
     shape = -2.4,   /* Illegal - enum values must be int */
     size = aConstant /* Legal */
  } attributes;
  struct{
      unsigned illegal_bitfield : h;  /* Illegal */
      unsigned legal_bitfield : size; /* Legal */
  }bits;
  int z[size];  /* Legal - array dimension is a constant */
  int w[3.14159];  /* Illegal - constant must be an int */
}
```

code segment too big

See **link failed**.

constant too large

The absolute value of decimal integer constants must be less than or equal to 2147483647 ($2^{31}-1$). Hexadecimal integer constants must range from 0x0 to 0xffffffff inclusive. If you want to express an unsigned number larger than 2147483647, you must express it as a hexadecimal number. Example:

```
unsigned long a = 2147483648;  /* Illegal */
unsigned long b = 0x80000000;  /* Legal - equals 2147483648 */
unsigned long c = 0x100000000; /* Illegal - larger than 32 bits */
```

continue outside of loop

The `continue` statement must be inside a loop (`while`, `do-while`, `for`). Example:

```
function()
{
  int i;
  continue;                /* Illegal */
  while(1)
      continue;            /* Legal */
  do { continue; }         /* Legal */
      while(1);
  switch(i){
  case 1:
      continue;            /* Illegal */
  }
}
```

data segment too big

See **link failed**.

debug table overflow

You've exceeded the size limits for debugging information tables. You can usually fix this by using a precompiled header. If you're not using MacHeaders, consider using it. If you don't want to use MacHeaders, make a precompiled header of the header files you are using. See Chapter 10 to learn how to make precompiled headers.

declarator too complex

The declaration statement is too complex. Example:

```
int **************************x[1][1][1][1][1][1][1][1][1][1][1][1];
```

default not in switch

The `default` keyword was found outside of a `switch` block.

duplicate case

A `case` constant expression evaluates to the same value as a previous `case` constant expression within the same `switch` block.

duplicate default

There may only be one `default` specified in a `switch` block.

expression too complex

An expression was too complex. Try breaking up the expression into subexpressions. Example:

```
typedef int ****************************** indirect_type;

function(meta_ptr)
indirect_type ******************************meta_ptr;
{
   int i;
   indirect_type intermediary;
   /* The following expression is too complex */
   i = ****************************
       **************************** meta_ptr;
   /* The equivalent broken up into two subexpressions */
   intermediary = ******************************meta_ptr;
   i = **************************** intermediary;
}
```

floating-point expression too complex

The compiler has used up all of its 68881 floating point registers. Break up the floating point expression or use fewer floating point register variables.

formal parameter *'symbol'* appears more than once

The formal parameter *symbol* was used in the function parameter list more than once. Example:

```
function(again, again, again)  /* Illegal */
int again;
{
}
```

function definition does not match prototype

The data types in a prototype defined for a function don't match the types in the definition of the function. Example:

```
extern function(char);  /* the prototype */
...
function(c)
int c;              /* type doesn't match prototype */
{
}
```

"function returning array" is not allowed

A function cannot return an array. However, a function can return a pointer that is the address of an array. Example:

```
char function_returning_array()[SIZE];        /* Illegal */
char *function_returning_ptr_to_array();      /* Legal */
```

"function returning function" is not allowed

A function cannot return a function. However, a function can return a pointer to a function. Example:

```
int function_returning_function()();    /* Illegal */
typedef int (* function_ptr)();
function_ptr  f_returning_ptr_to_f();   /* Legal */
```

Identifier does not match #ifdef

The `#endif` and `#else` macro preprocessor statements take an optional argument which is the symbol that the matching `#ifdef` or `#ifndef` (but not `#if` or `#elif`) used. If the optional argument is present, it must match the corresponding `#ifdef` or `#ifndef`. Example:

```
#ifdef macro_name
#else  Other_name /* Illegal - identifier does not match */
#endif Other_name /* Illegal - identifier does not match */
#ifdef macro_name
#endif macro_name /* Legal */
```

Illegal #else

Every `#else` or `#elif` must have a matching `#if`, `#ifdef`, or `#ifndef`. Example:

```
#ifdef name
#else
#else            /* Illegal - has no matching #ifdef */
#endif name
#elif condition  /* Illegal - has no matching #if */
```

Illegal #endif

Every `#endif` must have a matching `#if`, `#ifdef`, or `#ifndef`.

Illegal array bounds

The bounds of an array can't be negative or zero. Arrays can't contain over 32767 bytes. Examples:

```
int a[-7];       /* Illegal - bounds can't be negative */
int b[0];        /* Illegal - bounds can't be zero */
char c[32768];   /* Illegal - total array size > 32767 bytes */
int d[16384];    /* Illegal - total array size > 32767 bytes */
long e[8192];    /* Illegal - total array size > 32767 bytes */
```

Illegal cast

It is illegal to cast `structs` or `unions` to other types. It is legal to cast numerical values or pointers to other numerical values/pointers. Example:

```
typedef struct {int v, h;} Point;
function()
{
  Point p;
  long l;
  p = (Point) l;      /* Illegal */
  p = *(Point *) &l   /* Legal */
}
```

Illegal floating-point operation

You cannot use some of the C operators on floating point values. Example:

```
function()
{
  double d1;
  double d2;
  double d3;
  d3 = d1 % d2;    /* Illegal - can't do floating modulo */
  d3 = d1 << 3;    /* Illegal - can't do floating shifts */
  d3 = 3 << d1;    /* Illegal - can't do floating shifts */
}
```

Illegal function prototype

Only types and optional identifiers are allowed in prototypes. Example:

```
extern  char *badProto(int, char, cem[8]);  /* Illegal */
extern  int naughtyProto(int p, char fn());       /* Illegal */
```

You will also get this error if you try to mix "new style" function definitions with "old style" definitions. Example:

```
MixedFunc (int x, y)
    int y;
{
    ...
}
```

Illegal operation on array

You cannot use many of the C operators on arrays. Example:

```
function()
{
  char a1[4];
  char a2[4];
  char a3[4];
  int i;
  if ( a1 == a2 )   /* Legal - compares arrays' addresses */
   a1++;            /* Illegal - can't increment arrays */
  a1 = 0;           /* Illegal - can't assign to an array */
  a1 = a2;          /* Illegal - can't assign arrays */
  i = a1 - a2;      /* Legal - subtracts arrays' addresses */
}
```

Illegal operation on function

You cannot use many of the C operators on functions. Example:

```
function()
{
  int f1();
  int f2();
  int f3();
  int i;

  if (f1 == f2)   /* Legal - compares functions' addresses */
    f1++;         /* Illegal - can't increment functions */
  f1 = 0;         /* Illegal - can't assign to a function */
  f1 = f2;        /* Illegal - can't assign functions */
}
```

Illegal operation on struct/union

You cannot use many of the C operators on structs/unions. Example:

```
int function()

  struct{
    int x;
  }s1, s2, s3;
  int i;

  if (s1 == s2) /* Illegal - can't compare structs */
    s1++;       /* Illegal - can't increment structs */
  s1 = 0;       /* Illegal - can't assign int to a struct */
  s1 = s2;      /* Legal - struct assignment is allowed */
  s3 = s1 + s2; /* Illegal - can't add structs */
  i = s1 - s2;  /* Illegal - can't add structs */
  i = (long)s1; /* Illegal - can't cast a struct */
  return(s1);   /* Illegal - return does an implicit cast */
}
```

Illegal operator

(Debugger only) You've entered an expression with an operator that has a potential side effect: a function call, ++. or --.

Illegal pointer/integer combination

Integers may not be assigned to pointers with the exception of the constant zero. In addition, pointers may not be compared with integers, again with the exception of the constant

zero. In both cases a cast can be used to force the assignment or comparison where necessary. Example:

```
function()
{
  int *int_ptr;
  int i;
  int_ptr = 0x220;          /* Illegal */
  int_ptr = 0;              /* Legal */
  int_ptr = (int *) 0x220; /* Legal - address of MemErr */
  i = * 0x220;              /* Illegal */
  i = * (int *) 0x220;      /* Legal - contents of MemErr */
  return(int_ptr);          /* Illegal - implicit cast */
}
```

Illegal pointer arithmetic

The arithmetic being performed on the pointer is illegal. Example:

```
function()
{
  char *p1, *p2;
  long result;
  result = p1 + p2;  /* Illegal - can't add pointers */
  result = p1 / p2;  /* Illegal - can't divide pointers */
}
```

Illegal return type for pascal function

Pascal functions are allowed to return only `void`, integers, and pointers. Example:

```
typedef struct {int v, h;} Point;
pascal Point /* Illegal - can't return a Point */
pascal_function(x,y)
{
  Point p;
  p.v = x;
  p.h = y;
  return p;
}
pascal double illegal_pascal_fnc(); /* Illegal return type */
pascal void pascal_proc();          /* Legal */
```

Illegal size for bitfield

Bitfield sizes must be less than or equal to the number of bits in the word type. (See Chapter 16, § 8.5.) Labelled bitfield sizes must be greater than zero. Unlabeled bitfield sizes equal to 0 force a word alignment. Example:

```
struct bitfields{
  char c : 9;      /* Illegal - chars have 8 or fewer bits */
  int i : 17;      /* Illegal - ints have 16 or fewer bits */
  int zero : 0;    /* Illegal - bitfield size must be > 0 */
  int good : 11;   /* Legal */
  int : 0;         /* Legal - this forces word alignment */
  long l : 33;     /* Illegal - longs have <= 32 bits */
  int z : -4;      /* Illegal - bitfield size must be > 0 */
};
```

Illegal size for this operation

(Assembly) The size specifier for an assembly language instruction is the wrong size for what you're trying to do. Example:

```
asm {
    move.b  a0,a1    /* can't move bytes to address registers */
}
```

Illegal token

Certain characters that are part of the ASCII character set and all of the characters in the extended Macintosh character set are illegal tokens in THINK C. The character # is also an illegal token when it is other than the first character in a line preceded by any number of whitespace characters. However, any character can occur within comments, string literals or character literals. Example:

```
int a$variable;         /* Illegal */
char c = '$';           /* Legal to have $ here */
#define N    "###"      /* Legal to have # here */
char d = #define M 4;   /* Illegal */
```

Illegal type for bitfield

The only allowed types for a bitfield are `char`, `short`, `int`, and `long`, and these types prefixed by the modifier `unsigned`. Any other type for a bitfield is illegal. Example:

```
struct bitfields{
  char c : 5;            /* Legal */
  unsigned int i : 5;    /* Legal */
  long l : 17;           /* Legal */
  double d : 10;         /* Illegal */
  void *p : 8;           /* Illegal */
};
```

Illegal use of class type

A variable of class type must be declared as a pointer. Example:

```
function()
{
    ClassName   badObject;      /* Illegal */
    ClassName   *goodObject;    /* Legal */
}
```

Illegal use of inline function

Even though inline functions and Toolbox routines look like function calls, they are not. Hence, it is illegal to take the address of an inline function or a Toolbox routine. Another possible source of error is forgetting to put in the parameter parentheses in a built-in call that takes no arguments. Example:

```
/* PrOpenDoc is defined in PrintTraps.h */
pascal TPPrPort PrOpenDoc()     = { 0x2F3C, 0x0400, 0x0C00, 0xA8FD };

function()
{
  void *generic_pointer = FrameRect; /* Illegal */
  void *another_pointer = PrOpenDoc;

  /* what a forgetful Pascal programmer might write */
  while (Button)       /* Illegal - () missing */
    ;
  while(Button()) /* Legal */
    ;
}
```

Illegal use of type name *'symbol'*

A `typedef` name has appeared where it shouldn't. If a `typedef` name is used instead of a variable name, then an error occurs. Example:

```
typedef char Byte;
function()
{
  return Byte;      /* Illegal */
}
```

Illegal use of void

You cannot use most of the C operators on `void` values. Example:

```
void f()
{
  int i, j;
  i = (void)j + 5; /* Illegal - can't add a void */
  i = f() - 3;     /* Illegal - f() returns void */
}
```

Immediate operand out of range

(Assembly) Some assembly language instructions must be a certain size. Example:

```
asm {
    addq        #10, d0     /* immediate limit is 1-8 */
}
```

Incomplete macro call

A preprocessor command was found while reading macro parameters during macro expansion. Preprocessor commands are not allowed in this situation. Example:

```
#define macro(arg1, arg2) (arg1 > arg2)
macro(x,
#ifdef AndIfYouCallNow    /* Illegal */
1
#else
2
#endif
)
```

Incorrect base register

(Assembly) You're probably trying to reference a global off of a register other than A5 (for applications) or A4 (for desk accessories, device drivers, or code resources). You don't need to provide a base register to use a global variable in assembly language. Example:

```
int theGlobe;    /* a global */

OffBase()
{
    asm {
        move    theGlobe(a2), d0      /* Incorrect */
        move    theGlobe(a5), d0      /* Correct, but unnecessary */
        move    theGlobe, d0          /* Correct, and much better */
    }
}
```

Initialized object too complex

Initialization of a deeply nested `struct` will cause this error. Example:

```
struct s1{
struct s2{
...
struct s21{
    int i;
}...}}s = {4};        /* Too deeply nested */
```

Initialization to an address is illegal in a non-application

If the THINK C project is a code resource, desk accessory, or device driver, it is illegal to initialize an address in global or static memory. Example:

```
static int i;
static int *p = &i;  /* Illegal in non-application */
```

Invalid class definition

You didn't define a class correctly. Example:

```
union MyClass : indirect {     /* Illegal. Union can't be a class. */
    int     AnIVar;

    int     AMethod(int x);
};

struct MyClass : indirect {     /* Legal. */
    int     AnIVar;

    int     AMethod(int x);
};
```

Invalid declaration

Example:

```
/* Illegal - only a defining instance can be initialized */
extern int x = 1;
extern int y;
int y = 1;          /* Legal */
```

Invalid function definition

Example:

```
/* Illegal - function definition can't be declared extern */
extern f1()
{
}
f2() = 2;  /* Illegal */
```

Invalid method call

A method call doesn't make sense in this context. Example:

```
AMethod(int x)                /* Not a method definition */
{
    inherited::AMethod(x);  /* invalid method call */
}

AnObject::ARealMethod(int x)     /* A real method definition */
{
    inherited::ARealMethod(int x);    /* Legal */
}
```

Invalid redeclaration of *'symbol'*

An identifier was declared twice, and this redeclaration is incompatible with the first. Example:

```
extern int x;
long x;     /* Illegal - x is redeclared differently */
extern int y;
int y;      /* Legal - y is redeclared as the same type */
int z;
int z;      /* Illegal - can only have one defining instance */
typedef int IsARose;
IsARose IsARose;    /* Illegal */
```

Invalid register list

(Assembly) The register list contains duplicates, or the list is in the wrong order. Example:

```
asm {
    movem   a0/a0,-(sp)
    movem   d7-d1,-(sp)
}
```

Invalid storage class

Incompatible attributes have been explicitly or implicitly applied to a data item. Example:

```
register int aGlobal;  /* Illegal - register global */
extern auto int y;     /* Illegal - contradictory storage class */
main()
{
}
```

Invalid type

The specified combination of type keywords create an ambiguous or impossible data item. Example:

```
unsigned double d;          /* Illegal */
long struct {int i;} s;      /* Illegal */
```

jump table too big

See **link failed**.

label *'symbol'* already defined

The label *symbol:* has been defined twice within the function.

label *'symbol'* not defined

There was a `goto` to the label *symbol*: but none has been defined within the function.

link failed

The linker was unable to link the program. Usually the link fails due to an undefined symbol or to a multiply defined symbol (a symbol defined more than once in a project). To find out which files contain the offending symbols, use the **Check Link** command in the **Project** menu. The Link Errors window will display the names of the files in parentheses.

- code segment too big — one or more of your code segments has exceeded the 32K limit. See "Segmentation" in Chapter 7.
- data segment too big — you've declared more than 32K of global and static data in your project. Use memory allocation to create large data structures.
- jump table too big — you've exceeded the 32K jump table limit
- multiply defined: 'symbol' — 'symbol' was defined more than once
- resource too big — for drivers, DRVR + DATA resource is > 32K; for multi-segment drivers, DATA + JUMP > 32K; for code resources, CODE + DATA + JUMP > 32K; for multi-segment code resources, CODE + DATA + JUMP > 32K for segment containing `main()`, CODE > 32K for other segments.
- undefined: 'symbol' — there was a reference to 'symbol' that was never defined

lvalue required

An lvalue is an expression that refers to an object in memory that can be stored to, as well as examined. Example:

```
int int_f();
int *pint_f();
function()
{
  int i;
  int *p = &i;
  /* operand of ++ must be an lvalue */
  7++;              /* Illegal */
  int_f()++;        /* Illegal */
  pint_f()++;       /* Illegal */
  /* left operand of an assignment must be an lvalue */
  int_f() = i;      /* Illegal */
  pint_f() = i;     /* Illegal */
  *pint_f() = i;    /* Legal - * produces an lvalue */
  (*p)++;           /* Legal */
  (*pint_f())++;    /* Legal */
}
```

macro name already #define'd

It is illegal to `#define` a macro name that has already been `#define`'d. However, it is always legal to `#undef` a macro name whether or not it has been `#define`'d before. If you use a `#undef` before a `#define`, you will be sure that a subsequent `#define` will always work. Example:

```
#define macro_name 1
#define macro_name 2    /* Illegal - already #define'd */
#undef macro_name
#define macro_name 3    /* Legal to #define AFTER an #undef */
```

macro parameter *'symbol'* appears more than once

The formal parameters in a macro definition must appear only once. Example:

```
#define macro(again,again)   (again+7)   /* Illegal */
```

memory critical - proceed at your own risk

Now would be a really good time to quit. Save your files!

missing #endif

Every `#if`, `#ifdef`, and `#ifndef` must have a matching `#endif`.

missing '('

There is a missing parenthesis. `if`, `while`, do-`while`, `switch`, and `for` statements require the expressions following them to be contained inside parentheses. Example:

```
function()
{
  int flag, value, i;
  if (flag) return 1;          /* Legal */
  if flag return 2;            /* Illegal */
  while flag {...}             /* Illegal */
  do {...} while flag;         /* Illegal */
  switch value {...}           /* Illegal */
  for i = 0; i < 10; i++ {...} /* Illegal */
}
```

missing ')'

There is a missing right parenthesis. Example:

```
function(i)
int i;
{
  if (i > 4     /* Illegal - missing close parenthesis */
    /* then clause */
}
```

missing ':'

There is a missing colon in a conditional (`?:`) expression or in a `case` or `default` label. Example:

```
function(flag)
int flag;
{
  int i;
  i = flag ? 3 4; /* Illegal - Missing colon */
  switch(i){
  case 3          /* Illegal - Missing colon */
    return i+5;
  default         /* Illegal - Missing colon */
    return i+6;
  }
}
```

missing ';'

A semicolon was expected, but one was not found. Sometimes it is not possible for THINK C to determine that a semicolon was missing which will result in the error message **syntax error**.

missing ']'

A closing bracket was expected to match an open bracket, but one was not found.

missing '{'

To initialize a struct or an array, you need to enclose the initializers within braces. Example:

```
Rect myBadRect = 1;                    /* Illegal */
Rect myGoodRect = { 1, 2, 19, 61 };    /* Legal */
```

multiply defined: *'symbol'*

See **link failed**.

no files in project

You need at least one file in the project window for what you're trying to do.

no members defined

A `struct` or `union` was defined without members. Example:

```
struct memberless_struct{ };     /* Illegal */
union memberless_union { };      /* Illegal */
```

no methods defined

A class definition must have at least one method.

```
struct BadName : indirect {
    int     instanceVar;

    /* Illegal. No methods declared. */
};
```

out of *memory*

THINK C ran out of memory. Close open windows to reclaim more memory. You might want to check the More Memory option in the Preferences section of the **Options...** command in the **Edit** menu.

If you get this message when you try to run the debugger, try making application's partition size smaller. See "Memory Considerations" in Chapter 11.

pascal argument wrong size

The call to a built-in Toolbox or OS function has a non-integral argument of the wrong size. Example:

```
function()
{
  Rect r;
  Point p;
  ...
  FrameRect(r);      /* Illegal - Should pass ptr to Rect */
  FrameRect(&r);     /* Legal - Correct call to FrameRect */
  FrameRect(4);      /* Legal but wrong - 4 gets cast to ptr */
  SetPt(&p,10L,3);   /* Legal - 10L gets cast to an int */
  SetPt(p,4,5);      /* Legal but wrong - sizeof(p) == sizeof(&p)*/
  SetPt(&p, &p, 5); /* Illegal - ptr won't be cast to int */
  SetPt(&p, p, 5);  /* Illegal - struct won't be cast to int */
}
```

pointer required

A pointer was implied by an operator, but there is no pointer. Example:

```
function()
{
  int x, result;
  result = *x;            /* Illegal - x is not a pointer */
  result = x->a_member;   /* Illegal - x is not a pointer */
}
```

pointer types do not match

Incompatible pointers were used in an assignment, comparison, or subtraction. In THINK C, pointers are more strictly typed than some other C compilers. In assignments and comparisons, the two pointer types must match or be of type `void *`. When subtracting two pointers, the types must match and not be `void *`. Of course, pointers may be cast to force compatibility. Example:

```
function()
{
  char *char_ptr;
  int *int_ptr;
  void *void_ptr;
  int result;
  long difference;
  char_ptr = int_ptr;                       /* Illegal */
  char_ptr = (char *)int_ptr;               /* Legal */
  void_ptr = int_ptr;                       /* Legal */
  int_ptr = void_ptr;                       /* Legal */
  result = (char_ptr == int_ptr);           /* Illegal */
  result = ((int *)char_ptr == int_ptr);    /* Legal */
  result = (char_ptr == (void *)int_ptr);   /* Legal */
  result = (void_ptr == int_ptr);           /* Legal */
  difference = char_ptr - int_ptr;          /* Illegal */
  difference = char_ptr - (char *)int_ptr;  /* Illegal */
}
```

prototype required for '*symbol*'

When the Require Prototypes option in the Compiler Flags section of the **Options...** dialog is checked, you must provide a function prototype for each of your functions.

recursive #include or preprocessor overflow

The most likely cause of this error is that an `#include` file has `#included` itself directly or indirectly. Another possibility is deeply nested `#ifdefs` or macro invocations. Example:

```
#define name_1     0
#define name_2     name_1
...
#define name_54    name_53
int i = name_54;
/* Deeply nested macro overflows preprocessor */
#ifdef name_1
#ifdef name_2
...
#ifdef name_54
/* Deeply nested #ifdef overflows preprocessor */
```

redefinition of existing struct/union/enum

A `struct`, `union`, or `enum` with the same tag was declared twice. Only one defining instance is allowed. `struct`, `union`, and `enum` tags share the same name space. Example:

```
struct Rumpelstiltskin{
  int member;
};
/* The following are illegal only because the tag    */
/* Rumpelstiltskin has been used previously          */
/* Illegal */
struct Rumpelstiltskin {
  int member;
};
/* Illegal - union name conflicts with previous struct name */
union Rumpelstiltskin {
  void *where_prohibited;
  long l;
};
/* Illegal - enum name conflicts with previous struct name */
enum Rumpelstiltskin {
  Jakob_Ludwig_Karl_Grimm,
  Wilhelm_Karl_Grimm
};
```

required array bounds missing

No size was specified for an array when one was needed to determine data storage space. The size can be explicit or implicit through the use of an initializer. Example:

```
extern char a[];    /* Legal - bounds not needed */
int i = sizeof(a); /* Illegal - need to know size */
function(b)
char b[];           /* Legal - bounds not needed */
{
  char c[];         /* Illegal - auto array needs bounds */
}
```

resource too big

See **link failed**.

stack frame too large

The local and temporary variable stack space required by a function exceeded 32768 bytes.

statements nested too deeply

THINK C allows nesting of at least 20 statements. `else if`s do not introduce nested `if` statements and can be put together in arbitrarily long chains. Example:

```
/* maximum nesting is 20 */
if (condition_1) … if (condition_20) {/* then clause */}

/* arbitrary number of else-ifs can be chained */
if (condition_1)          {/* then clause 1 */}
else if (condition_2)     {/* then clause 2 */}
...
else if (condition_many) {/* then clause many */}
```

struct/union too large

The declared `struct/union` exceeded 32768 bytes of data storage.

switch value must be integral

A switch expression must be of type `char`, `int`, or `long` or an `unsigned` variant. Example:

```
char *p;
float f;
switch(p){…}      /* Illegal - switch of pointer */
switch(f){…}      /* Illegal - switch of float */
```

syntax error

There was a syntax error. Some common errors: too many } s, label without :, malformed expressions, and of course, a missing semi-colon.

there are no void objects!

Declarations provide storage space for the variable declared. It is an error to have a variable that has no storage space. It is legal to have a pointer to `void`; this is a C idiom for a generic pointer. Example:

```
void nugatory;          /* Illegal */
void *generic_pointer; /* Legal */
```

too many formal parameters

THINK C allows you to have up to 25 formal parameters in a function definition. Example:

```
f(arg1, arg2..., arg25, arg26)  /* one too many args */
{
}
```

too many initializers

The number of initialization values exceeds the expected number of data items specified in the declaration of the data structure. Example:

```
char *directions[4] =
    {"north", "east", "south", "west", "lost" }; /* Illegal */
struct { int a,b,c; } x[2] = { 1, 2, {3, 4 } };  /* Illegal */
```

too many macro parameters

Macros are allowed to have up to 25 arguments. Example:

```
#define macro(arg1, arg2, ..., arg26)  /* one too many args */
```

too many segments

Applications can have no more than 254 segments. Multi-Segment desk accessories, device drivers, and code resources are limited to 30 segments.

undefined enumeration

An enumeration declaration refers to an enumeration tag that has not been defined. Example:

```
enum unknown colors;  /* Illegal */
```

undefined struct/union

A `struct/union` must be defined before an instance of it can be declared. However, a pointer to an undefined `struct/union` is legal since the size of the pointer is known and the size of the undefined `struct/union` is not needed. Example:

```
struct not_previously_defined s;          /* Illegal */
struct not_previously_defined *p;         /* Legal */
int i = sizeof(p);                        /* Legal */
int j = sizeof(*p);                       /* Illegal */
struct link_list_element{
  struct link_list_element *next;         /* Legal */
  struct link_list_element recursive;     /* Illegal */
};
```

undefined '*symbol*'

See **link failed**.

unexpected end-of-file

End of file was reached before a C language construct was completed. Example:

```
main(
/* EOF - end of file encountered before close parenthesis */
```

unions may not have bitfields

`unions` may not have bitfields. However, they may include `structs` that have bitfields. Example:

```
union {
  int i;
  unsigned int bits : 5;    /* Illegal */
}union_1;
typedef struct { unsigned int bits : 5; } bitfield_type;
union {
  int i;
  bitfield_type b;          /* Legal */
}union_2;
```

unknown instruction

(Assembly) The inline assembler doesn't recognize the mnemonic you've provided. This usually happens when you forget the period in an mnemonic. Example:

```
asm {
    movew    d0,d1
}
```

unknown struct/union member *'symbol'*

In a `.` or `->` expression either the left operand was not a `struct/union`, or the right operand was not the name of a member of the type of the left operand. Example:

```
function()
{
  int non_struct, *int_ptr;
  struct s1_struct{ int member_1; }s1, *p1;
  struct s2_struct{ int member_2; }s2, *p2;
  /* These are all illegal */
  s1.non_member;        /* non_member is not in s1_struct */
  p1 -> non_member;     /* non_member is not in s1_struct */
  non_struct.member_1; /* non_struct is not a struct */
  int_ptr -> member_1; /* int_ptr is not a struct pointer */
  s1.member_2;          /* member_2 is not in s1_struct */
}
```

unterminated comment

End of file was reached before end of comment was detected.

unterminated quote

Either a character constant or a string constant is missing its end quote. Example:

```
function()
{
  long file_type;
  /* Illegal - missing " after world */
  function("hello world);

  /* Illegal - missing ' after TEXT  */
  file_type = 'TEXT;
  /* Legal */
  file_type = 'TEXT';
}
```

use of struct/union/enum does not match declaration

A `struct/union/enum` tag was used in a declaration that conflicts with the original `struct/union/enum` tag declaration. `struct/union/enum` tags share the same name space. Example:

```
struct Rumpelstiltskin {
  int member;
};
union Rumpelstiltskin anIllegalUnion;  /* Illegal */
enum Rumpelstiltskin anIllegalEnum;    /* Illegal */
```

void function must not return a value

A function declared void cannot return a value. Use a `return` statement without a value. Example:

```
void in_partners_suit(condition)
int condition;
{
if (condition)
  return;    /* Legal */
else
  return(1); /* Illegal */
}
```

wrong number of arguments to *'symbol'*

A call to the built-in Macintosh Toolbox or OS function *symbol* was made with the wrong number of parameters. Example:

```
SetPt(&aPoint, 3, 4, 8);  /* one too many arguments */
```

wrong number of arguments to macro *'symbol'*

A macro was called with the wrong number of arguments. Example:

```
#define twice(x)  (x+x)
function()
{
  int i;
  i = twice(3,4); /* Illegal - macro called with 2 args */
}
```

wrong number of operands

(Assembly) You've probably forgotten something. Example:

```
asm {
    add.w    d0
}
```

wrong type(s) of operand(s)

(Assembly) Some instructions require operands of a specific type. Example:

```
asm {
    movea    d1,d0    /* movea can only move into address registers */
}
```

zero-sized object

An operator was used illegally on a zero-sized object. Example:

```
void vacuous()
{
  int i;
  void *nowhere, *erewhon;
  i = sizeof( vacuous() );     /* Illegal */
  i = nowhere - erewhon;       /* Illegal */
}
```

RMaker Reference
D

Introduction

Macintosh programs are designed around objects. Windows, menus, dialog boxes, and alerts are all objects that the Macintosh uses. You can build these objects on the fly in your programs, or you can load them in from the resource fork of your application. The advantage of storing these Macintosh objects as resources is that your program's function (the code) is separate from the user interface (the look).

RMaker is a resource compiler. It takes a textual specification of the objects to be used in a program and produces the resource data structures which are understood by the Macintosh Toolbox routines.

To learn how to use resources with THINK C projects, read the "Anatomy of a Project" section in Chapter 7. To learn about resources read *Inside Macintosh I*, Chapter 5, "The Resource Manager."

Topics covered in this appendix

- Using RMaker
- RMaker file format
- Predefined resource types

Using RMaker

To create a resource file, use the THINK C editor to create the RMaker source file. Then use the **Transfer...** command in the **File** menu to launch RMaker. RMaker displays a file selection dialog. Choose your RMaker source file, and RMaker will produce a resource file from it.

RMaker File Format

RMaker input is line-oriented and has a quite rigid syntax. The RMaker input file consists of an output specification followed by an arbitrary number of resource definitions separated by blank lines. Comments are also allowed, either as whole lines or as tags at the end of lines. Comment lines begin with an asterisk (`*`). Comments at the end of lines are preceded by two semicolons (`;;`).

All numbers in your file are decimal, except in certain resource type declarations (see below). To enter special characters, use a backslash followed by two hexadecimal digits. For example, the code for the Apple (🍎) symbol is `\14`. A ++ at the end of a line tells RMaker that the line is continued on the next line.

You can use the `/QUIT` directive to tell RMaker to quit after it builds your resource file. The `/NOSCROLL` directive tells RMaker not to scroll your file in its source window. For example:

```
/QUIT
/NOSCROLL
FileName
????SAMP

* resource declarations here...
```

Output file specification

RMaker files begin with the output specification. The name of the output file appears on the first line, followed by the file signature. The file signature is a four-character file type followed by a four-character creator. To learn more about file signatures, see *Inside Macintosh III*, Chapter 1, "The Finder Interface."

For example, if you want to name the output file `Sample` and give it the file type `????` and creator `SAMP`, the first two lines of the RMaker input file would be:

```
Sample      ;; the name of the output file
????SAMP        ;; the file type ???? and creator SAMP
```

The file signature line may be left blank, in which case the file type and file creator will be set to nulls (four bytes of 0 each). The output specification may be preceded by an arbitrary number of blank lines or comment lines.

If the file name begins with an exclamation point `(!)`, RMaker merges the resources into that file instead of creating a new file. In this case, if you don't supply a file signature, RMaker doesn't touch the file signature of the file you're merging into.

You can use the `INCLUDE` *`filename`* statement to merge the resources from another resource file into the output file.

Resource declarations

The body of an RMaker source file consists of groups of resource declarations headed by a resource type clause.

(Note: in the following examples, the brackets `[]` enclose optional data. Words in *italics* describe user-supplied data.)

A resource type section begins with a TYPE declaration:

TYPE *resource type* [= *resource type*]

The = clause lets you define your own own resource types. If the optional = clause is not present, the first resource type must be a predefined type. If the clause is present, the resource type named there must be a predefined type.

The resource declarations come after the TYPE declaration. They look like this:

[*name*], ID [(*attributes*)]
type-specific resource data

Note that the comma separating the name from the ID must be included even if you don't name the resource. The attributes byte must be surrounded in parentheses. See the *Inside Macintosh I*, Chapter 5, "The Resource Manager" to learn more about resource attributes.

A blank line must follow the resource definition.

Predefined Resource Types

RMaker recognizes these 12 resource types.

```
'ALRT' - Alert
'BNDL' - Bundle
'CNTL' - Control
'DITL' - Dialog (or Alert) Item List
'DLOG' - Dialog
'FREF' - File Reference
'GNRL' - General
'MENU' - Menu
'PROC' - Procedure (contains code)
'STR ' - String
'STR#' - String List
'WIND' - Window
```

The following sections illustrate the different types of resource declarations:

'ALRT' - Alert Template

```
TYPE ALRT
    , 128                ;; the resource number
70 100 150 412           ;; rect for the alert (top left bottom right)
10                       ;; the resource ID for the item list
FFFF                     ;; stages word (hex)

* always remember the blank line at the end of a resource
* definition - it is a required separator
```

'BNDL' - Bundle Template for Application

```
TYPE BNDL
    ,128                 ;; resource number
SAMP 0                   ;; creator for bundle
ICN#                     ;; resource type (icon list)
0 128 1 129              ;; local ID 0 maps to ICN# 128, 1 to 129
FREF                     ;; resource type (file reference)
0 128 1 129              ;; local to FREF mapping 0 to 128, 1 to 129
```

'CNTL' - Control Template

```
TYPE CNTL
    ,128                 ;; resource number
MyControl                ;; title for control
10 10 20 20              ;; rectangle for cntl (top left bottom right)
Visible                  ;; may also be Invisible
0                        ;; CDEF proc ID
0                        ;; reference constant (defines control type)
0 100 0                  ;; minimum value, maximum value, initial value
```

'DITL' - Dialog (or Alert) Item List

```
TYPE DITL
    ,10                  ;; resource number
9                        ;; number of items in the item list

button                   ;; enabled button items (enabled by default)
20 20 40 100             ;; rectangle (window-relative coordinates)
Cancel                   ;; text in the button

radioButton              ;; radio button item
50 20 70 120             ;; rectangle (includes button and text)
Push Me                  ;; the text goes to the right of the button

radioButton disabled     ;; dimmed radio button item
50 20 70 120             ;; rectangle (includes button and text)
```

```
Can't Push Me              ;; you can't push a disabled button

checkBox                   ;; check box item
80 20 100 120              ;; rectangle (includes check box and text)
Check Me                   ;; the text goes to the right of the box

staticText                 ;; static text item
20 120 40 320              ;; rect of text
This text would get placed next to the Cancel button ;; the text

editText Disabled          ;; disabled editable text item
50 140 90 320              ;; rect of the box for editable test
initial string             ;; initial edit text

editText                   ;; editable text item (enabled by default)
100 140 120 320            ;; rectangle
you can edit this text     ;; the initial string for editable item

iconItem                   ;; for the display of an icon
100 100 132 132            ;; rectangle should be 32x32
3                          ;; resource ID for icon (Type ICON)

picItem                    ;; to display of a Quickdraw picture
30 100 20 200              ;; display rectangle (picture will be scaled)
57                         ;; resource ID for picture (Type PICT)

userItem                   ;; a user-defined item
30 40 80 90                ;; the rectangle
```

'DLOG' - Dialog Template

```
TYPE DLOG
    ,128                   ;; the resource number
My Dialog Box              ;; a message
70 100 150 412             ;; the rectangle (top left bottom right)
Visible NoGoAway           ;; or Invisible or GoAway
0                          ;; the dialog definition ID
0                          ;; the refCon, available to the user
10                         ;; the resource ID for the dialog item list
```

'FREF' - File Reference

```
TYPE FREF
    ,128                   ;; the resource number
APPL 0                     ;; the file type and local ID
```

'GNRL' - General

'GNRL' is used to define your own resource types and define their format. The resource's format is constructed from "elements." The elements available are:

- `.P` Pascal string
- `.S` String without a leading length byte
- `.I` Decimal integer
- `.L` Decimal 32-bit integer
- `.H` Hexadecimal integer
- `.R` Read the given resource from the given file.
  R takes the arguments `filename  resource TYPE resource ID`

```
TYPE ICN# = GNRL          ;; define the type ICN#
    ,128                  ;; the resource ID
.H                        ;; hexadecimal data follows
0001 8000 0002 4000       ;; ICN#'s need 2 icons (icon and
0003 C000 0004 2000       ;; mask) of 32x32 bits each, or 32
...                       ;; lines of two longwords apiece.
FFFF FFFF FFFF FFFF
```

'MENU' - Menu

```
TYPE MENU
    ,10                   ;; the resource number (Menu ID)
MyMenu                    ;; the menu title
First Item                ;; the first menu item
Second Item /S            ;; the second menu item with Command-S
(Third Item               ;; the third item ("(" disables items)
(-                        ;; a gray line (this is item #4)
Fifth Item                ;; the fifth item
```

'PROC' - Procedure (contains code)

```
TYPE PROC
    ,128                  ;; the resource number
Filename                  ;; the code from this file will get placed
                          ;; in the resource
```

'STR ' - String

```
TYPE STR                  ;; spelled 'STR ' - trailing space  required!
    ,128                  ;; the resource number
My Wild Irish Rose        ;; the string assigned to the resource 128
```

'STR#' - String List

```
TYPE STR#
    ,128                ;; the resource number
2                       ;; the number of strings in the list
The First String
And the second string   ;; the two strings in the list
```

'WIND' - Window

```
TYPE WIND
    ,128                ;; the resource ID
My Window               ;; the window title
40 40 200 472           ;; the window rect (top left bottom right)
Visible GoAway          ;; or Invisible or NoGoAway
0                       ;; the window definition ID
0                       ;; refCon, a long word available to user
```

Index

Entries in **bold face** refer to menu commands. Entries in `typewriter face` refer to functions, methods, variables, keywords, or files.

68020 option 413
68881
    defining unary inline functions 126
68881 option 413
\newline 442
\p 438
__STDC__ 442
A.P.P.L.E. Co-op 8
AboutToPrint 354
    CDocument 304
    CEditText 309
    CPane 338
    CPanorama 346
abstract classes 198
Activate
    CControl 279
    CDesktop 290
    CDirector 297
    CEditText 309
    CScrollBar 362
    CSizeBox 370
    CView 388
    CWindow 396
ActivateWind
    CDirector 297
ADBS resource 85
Add
    CCluster 272
AddButton
    CRadioGroup 359
AddButtonID
    CRadioGroup 360
AddDirector
    CApplication 238
additive operators 440
AddMenu
    CBartender 246
AddResMenu 243
AddSubview
    CView 390
AddWind
    CDesktop 292
AdjustBounds
    CStaticText 374
AdjustCursor
    CDesktop 292
    CEditText 309
    CView 389
AdjustHoriz
    CPane 336
AdjustScrollMax
    CScrollPane 367
AdjustToEnclosure
    CPane 336
AdjustVert
    CPane 336
ANSI 437
ANSI libraries 133
ANSI-style function definitions 129, 130
APDA 7, 75
Append
    CList 322
Apple Numerics Manual 7
AppleTalk interfaces 122
AppleTalk routines
    calling 122
applications
    building 74-76
    component size limits 70
arguments
    converting from Pascal to C 120
    functions accepting a variable number 167-168

arithmetic conversion 439
arrays
displaying 54-57, 150
arrow keys 96
asm See: assembly language
asm 68000 { ... } 156
asm 68020 { ... } 157
asm 68881 157
asm { ... } 156, 157
assembly language 155
addresses 165
branch optimizations 160-161
C identifiers in 158
calling Toolbox routines from 162
directives 156, 164-165
dispatch tables 165
hexidecimal constants 163
instruction size 170
jumping to another function from 161
labels in 159
local storage 170
MC68000 156
MC68020 157
MC68881 157-158
mixing MC68000 & MC68020 code 156, 157
multiple entry points 162
multiple entry points into functions 161
order within operands 164
pc-relative offsets 165
referencing struct fields 159
returning from a function from 161
short branch error 160
using C labels in 160
using C statements in 160
using constants 170
using register variables 158
using registers 163
AssignIdleChore 263
ATPOpenSocket 127
Attach Condition 143, 434
auto mode 146
AutoScroll
CPanorama 346
Background Null Events 76
Balance 97
basic types 439
BeginTracking
CMouseTask 327
bit-field declarations 440
bless 187, 194
blessD 190, 194
blocks 271
bounds 343
bounds rectangle 211
breakpoints 47, 142
conditional 143
setting in other files 142
temporary 143
BringFront
CList 322
Brodie 131
bug column 136
bureaucrat 225
bugs, fixing 116
Build Code Resource... dialog 91, 423
Byte, Pascal type 127
C calling conventions 166-168
C calling sequence 166
C function entry 166-167
C function exit 167
C Traps and Pitfalls 6
C++ 193
C: A Reference Manual 6
CalcAperture
CPane 339
CalcFrame
CPane 338
CalcTERects
CStaticText 374
Calibrate
CScrollPane 367
call chain 140
callback routines
in drivers 79
calling conventions
C 166
Pascal 168-170

calling Macintosh Toolbox routines 119
calling sequence
    C 166
    Pascal 168-169
CallPascal 124
CallPascalB 124
CallPascalL 124
CallPascalW 124
ccommand 133
CDecorator 231
CDEF resource 85
cdev resource 85
CenterWindow
    CDecorator 288
CenterWithinEnclosure
    CPane 337
CFWDesktop 230, 289
ChangeName
    CFile 319
ChangeSize
    CControl 279
    CPane 336
    CScrollPane 367
    CWindow 397
ChangeStation
    CRadioGroup 360
Char, Pascal Type 127
character constants 438
    four-byte long 121
Check Link 471
Check Pointer Types 129, 414
Check Syntax 116
CheckAllocation
    CError 314
CheckMarkCmd
    CBartender 247
CheckOSError
    CError 314
CheckResource
    CError 314
Chernicoff, Stephen 7
ChooseFile
    CApplication 238
circular references 185
class 178
    membership 187
class hierarchy 197
classes
    declaring 184
    direct 189
Clear All Breakpoints 54
Clear Breakpoint 142
Close 99
    CDataFile 284
    CDirector 297
    CDocument 303
    CFile 318
    CWindow 395
Close All 99
CloseWind
    CClipboard 267
    CDirector 297
    CDocument 303
CODE component 69
Code Generation 445
code generation options 127
code profiler
    setting the option 128
code resourcea
    component size limits 70
code resources
    building 84-92
    component size limits 70, 86
    global data in 86-87
    headers 89
    literals in 87
    locking 88-89
    merging 91
    multi-segment 86, 91-92
    Object C in 86
    Quick Draw globals 87
    reentrant 89
    using libraries in 87-88
    using MacTraps 87
    using objects 183
    writing 86
Code segment too large 450
command line
    UNIX 133
command number 241
command numbers 215
Commands.h 215
compatibility,THINK C 2.0 & up 9

Compile 115
compiler options 129
compiling 115
    files in project 116
    files not in project 115
    fixing errors 116
CompuServe 8
conditional breakpoints 143
conditional compilation 442
Confirm Saves 99
`ConfirmClose`
    CDocument 303
const 439
Consulair 172
`Contains`
    CDesktop 292
    CPane 335
    CView 387
    CWindow 395
context 147
contexts 51, 57, 149
controlling execution 44, 144
`ConvertGlobal`
    CClipboard 269
`ConvertPrivate`
    CClipboard 269
coordinate systems 211
coordinates
    frame 211
    global 211
    window 211
`Copy`
    CObject 330
`CreateDocument`
    CApplication 237
`CreateNew`
    CFile 318
credit limit 219, 226
`cstr2dec` 123
`CtoPstr` 121
`CurApName` 152
current function 138, 140
current statement 138, 140
current statement arrow 138
cursor tracking 214
*cust*omer supportl 449
`DA main.c` 135
DATA component 69
Data segment too big 450
Data window 137, 139
`DataSize`
    CClipboard 268
`Dawdle` 236
    CBureaucrat 257
    CEditText 309
`DC` 156
`DC.B` 164-165
`DC.W` 165
`Deactivate`
    CControl 279
    CDesktop 290
    CDirector 297
    CEditText 309
    CScrollBar 362
    CSizeBox 370
    CView 388
    CWindow 396
`DeactivateWind`
    CDirector 298
debugger
    breakpoints 47, 142
    conditional breakpoints 143
    controlling execution 44-47, 144-146
    Data window 139
    displaying arrays 54-57, 150
    displaying structs 150
    editing expressions 147
    editing files 141
    entering expressions 147
    evaluating expressions 58, 149
    expressions 149
    formats 58, 148
    memory considerations 152
    modifying values 149
    quitting 59, 152
    searching 141
    Source window 138
    termporary breakpoints 143
    turning on 42, 136
    windows 137
`dec2str` 123
`delete` 186, 194

DeleteFromBar
CBartender 247
deselect button 50, 139
desk accessories (See also: drivers)
building 76-84
component size limits 70
event record pointer 78-79
using objects 183
Desk Manager 67
desktop 198
device drivers (See also: drivers)
building 76-84
component size limits 70
Device Manager 67
DIBadMount 378
direct 184, 189
direct classes 189
direct commands 196
directors 225
DisableCmd
CBartender 247
DisableMenu
CBartender 247
disk layout diagram 16, 114
DispatchClick
CDesktop 291
CView 388
CWindow 397
DispatchCursor 389
CDesktop 291
CWindow 397
Dispose
CBureaucrat 256
CCluster 272
CControl 277
CDesktop 290
CDirector 296
CDocument 301
CObject 330
CPane 334
CPrinter 354
CStaticText 372
CView 386
CWindow 394
DisposeAll
CCluster 272
DisposeItems
CCluster 272
Do
CTask 382
DoActivate
CSwitchboard 378
DoAutoKey
CApplication 232
CBureaucrat 256
CEditText 309
DoClick
CControl 280
CEditText 309
CScrollBar 362
CView 388
DoCommand
CApplication 233
CBureaucrat 257
CDirector 296
CDocument 302
CEditText 308
document 225
DoDeactivate
CSwitchboard 378
DoDiskEvent
CSwitchboard 378
DoForEach
CCluster 273
DoForEach1
CCluster 273
DoGoodClick
CButton 260
CCheckBox 262
CControl 280
CRadioButton 358
DoHorizScroll
CScrollPane 367
DoKeyDown
CApplication 232
CBureaucrat 256
CEditText 309
DoKeyEvent
CSwitchboard 378
DoKeyUp
CApplication 232
CBureaucrat 257
DoMouseDown

CSwitchboard 378
DoMouseUp
CDesktop 291
CSwitchboard 378
CView 388
DonePrinting
CDocument 304
CEditText 309
CPane 338
CPanorama 346
CStaticText 373
DoPageSetup
CPrinter 354
DoPrint
CPrinter 354
DoResume
CSwitchboard 379
DoRevert
CDocument 304
DoSave
CDocument 304
DoSaveAs
CDocument 304
DoSaveFileAs
CDocument 305
DoSuspend
CSwitchboard 378
DoThumbDrag
CScrollPane 367
DoThumbDragged
CControl 280
CScrollBar 362
double
functions returning 167
DoUpdate
CSwitchboard 378
DoVertScroll
CScrollPane 367
Drag
CWindow 396
DragWind
CDesktop 292
Draw
CBorder 254
CControl 279
CPane 337
CPicture 350
CScrollBar 362
CSizeBox 369
CStaticText 372
DrawAll
CControl 279
CPane 337
driver glue 78, 79
drivers 76
building 76-84
callback routines 79-80
closing 83
control 83
fields 81, 82
global data in 79-80
header fields 81-82
multi-segment 77, 84
using objects in 77
opening 82, 83
prime 83
Quick Draw globals 80
returning 83-84
status 83
trap intercept routines 79-80
using libraries in 80-81
using MacTraps 80
writing 78
ellipsis (...) 130
EnableCmd
CBartender 247
EnableMenu
CBartender 247
encapsulation 178
enclosures 198
EnclosureScrolled
CPane 336
EnclToFrame
CPane 339
EnclToFrameR
CPane 339
EndTracking
CMouseTask 327
enter button 50, 139
entry field 50, 139
enum 441
enumeration constants 438
enumerations 441
errors

fixing 116
event loop 235
event record pointer
in desk accessories 78-79
examining variables 50
Exit
CApplication 237
ExitToShell 59, 152
expression column 50
extern 441, 442
external declarations 441
ExtractCommands
CBartender 249
ExtractHierMenus
CBartender 249
f_LongJump 237
f_SetJump 237
file creator 68
file inclusion 442
File Manager data structures 122
file names 110
file signatures 68
file type 68
files
closing 98
creating 93
editing 96
opening 93
printing 98
saving 98
Find Again 100
Find in Next File 102
FindCmdNumber 215
CBartender 248
FindIndex
CList 323
FindItem
CCluster 272
FindItem1
CCluster 273
FindItemText
CBartender 248
FindLine
CStaticText 374
FindMacMenu
CBartender 248
FindMenuIndex
CBartender 248
FindMenuItem
CBartender 248
FindSubview
CView 390
Find... 100
FirstItem
CList 322
FirstSuccess
CList 323
FirstSuccess1
CList 323
FitToEnclFrame 365
CPane 337
FitToEnclosure
CPane 336
Fix2X 122
fixing errors 116
FKEY resource 85, 86
float
passed as an argument 441
floating constants 438
floating point arithmetic 117
floating point numbers
sizes of 132
floppy disks
installing THINK C on 14-15
fonts 98
formats 58, 148
Frac2X 122
frame 208, 332
frame coordinates 211
FrameToBounds
CPicture 350
FrameToEncl
CPane 339
FrameToEnclR
CPane 339
FrameToGlobalR
CPane 339
CView 391
CWindow 398
FrameToWind
CPane 339
FrameToWindR
CPane 339
function declarators 441

function definitions 441
function entry
C 166-167
Pascal 169
function exit
C 167
Pascal 169-170
function pointers
as arguments to Toolbox routines 123
function prototypes 130-131, 441
functions
ANSI-style definitions 129, 130
multiple entry points 161-162
returning struct, union, or double 167
with variable-length argument lists 167-168
gApplication 225, 228, 301, 399
gBartender 231, 246, 399
gClicks 401
gClipboard 231, 265, 266, 400
gDecorator 231, 287, 400
gDesktop 230, 289, 394, 399
gError 313, 400
Get Info... 69, 110
GetAperture
CDesktop 293
CPane 335
CView 387
CWindow 395
GetBounds
CDesktop 293
CPanorama 345
GetCaretTime 257, 309
GetCmdText
CBartender 247
GetData
CClipboard 269
GetDblTime 401
GetExtent
CPanorama 344
CPicture 350
GetFrame
CPane 335
CView 387
CWindow 395
GetFramePosition
CPanorama 344
GetFrameSpan
CPanorama 345
GetGlobalScrap
CClipboard 268
GetHomePosition
CPanorama 345
GetIdNumber
CRadioButton 358
GetInterior
CBorder 254
CScrollPane 367
CView 387
CWindow 395
GetItemIcon 127
GetItemMark 127
GetLength
CDataFile 284
GetLengths
CPane 335
GetMacPicture
CPicture 350
GetMacPort
CView 387
GetMark
CDataFile 284
GetMaxValue
CControl 278
GetMinValue
CControl 278
GetName
CDocument 305
CFile 318
GetNameIndex
CTask 382
GetNextEvent 379
GetNumItems
CCollection 275
GetOrigin
CPane 335
CView 387
GetPageInfo
CPrinter 354

GetPixelExtent
CPane 335
CPanorama 345
GetPosition
CPanorama 345
GetPrintRecord
CPrinter 354
GetScaled
CPicture 350
GetScales
CPanorama 345
GetStation
CRadioGroup 360
GetStationID
CRadioGroup 360
GetSteps
CScrollPane 367
GetSupervisor
CBureaucrat 256
GetTEFontInfo
CStaticText 374
GetTextHandle
CStaticText 373
GetTitle
CControl 278
CWindow 395
GetTopWindow
CDesktop 293
GetValue
CControl 278
GetWCount
CDecorator 288
gGopher 199, 225, 256, 296, 301, 400
gHasWNE 400
gIBeamCursor 401
gInBackground 236, 401
gLastMouseDown 378, 401
gLastMouseUp 401
gLastViewHit 401
global coordinates 211
global data 79, 86
global scope 51, 58, 147
Global.h 244
globals, low memory 125
Go 49, 144, 431
Go Until Here 141, 146, 432
gopher 199
grep 103
GrowMemory 314
CApplication 234
GrowZoneFunc 234
CError 314
gSleepTime 257, 400
gUtilRgn 402
gWatchCursor 401
handles
in objects 191
Handles, dereferencing 126
Harbison 6
hard disk
installing THINK C on 13-14
header files 442
including only once 191
headers
once-only 110
hexidecimal
in assembly language 163
HFS Navigator 111
Hide
CControl 279
CDesktop 290
CPane 336
CView 387
CWindow 396
HideSuspend
CWindow 396
HideWind
CDesktop 292
HiShort 244
HitSamePart 401
CDesktop 292
CView 388
How to Write Macintosh Software 7
Human Interface Guidelines 7
IApplication
CApplication 229
IBartender 217
CBartender 246
IBorder
CBorder 253
IBureaucrat
CBureaucrat 256

IButton
CButton 260
ICheckBox
CCheckBox 262
IClipboard
CClipboard 267
ICluster
CCluster 272
ICollection
CCollection 275
icons 68, 74
IDataFile
CDataFile 284
IDecorator
CDecorator 288
identifiers 437
capitalization 116
length 116
IDesktop
CDesktop 290
IDirector
CDirector 296
Idle
CApplication 236
IDocument
CDocument 301
IEditText
CEditText 308
IFile
CFile 317
Ignore Case option 100
IList
CList 321
IMouseTask
CMouseTask 326
implicit variable (this) 188
In 145, 431
Includes
CCluster 272
indenting 97
indirect 184
inheritance
methods 189
inherited 189
INIT resource 85
InitMemory
CApplication 230
InitToolbox
CApplication 230
inline assembler See: assembly language
inline functions
defining 125
defining 68881 unary 126
InsertAfter
CList 322
InsertAt
CList 322
InsertHierMenu
CBartender 247
InsertMenuCmd
CBartender 247
Inside Macintosh 6
installing THINK C
on a hard disk 13-14
on floppy disks 14
InstallPanorama
CScrollPane 366
instance 178
instance variables 178, 187
address of 191
int
sizes of 132
integer constants 437
integral promotion 439
IPane
CPane 333
IPanorama
CPanorama 344
IPicture
CPicture 349
IPrinter
CPrinter 353
IRadioButton
CRadioButton 357
IRadioGroup
CRadioGroup 359
IsActive
CDirector 298
CView 387
IsChecked
CCheckBox 262
IScrollBar
CScrollBar 362

IScrollPane
    CScrollPane 366
IsEmpty
    CCollection 275
IsFloating
    CWindow 395
ISizeBox
    CSizeBox 369
IStaticText
    CStaticText 372
IsVisible
    CView 387
ISwitchboard
    CSwitchboard 377
ITask
    CTask 382
itsGopher 307
itsSwitchboard 377
IView
    CView 386
IViewRes
    CView 253, 308, 334, 344, 349, 366, 372, 386
IViewTemp
    CBorder 254
    CPane 334
    CPanorama 344
    CPicture 350
    CScrollPane 366
    CStaticText 372
    CView 386
IWindow
    CWindow 394
jIODone 83
JSR 161
JUMP component 69
JumpToEventLoop 314
    CApplication 237
KeepTracking
    CMouseTask 327
Kernighan 6, 131
keywords 437
    for objects 193
Knaster, Scott 7
LastItem
    CList 322
LastSuccess
    CList 323
LastSuccess1
    CList 323
LDEF resource 85
libraries 171
    in code resources 87
    in drivers 80-81
    moving 112
    THINK C 2.0 & up compatibility 9
limits
    project component sizes 70, 71
line splicing 442
Link Errors window 408
LLastClick 122
Load Library 171
local scope 51, 58, 147
Lock 58, 149
LoShort 244
low level debuggers 151
low memory globals 125
MacHeaders 117
    editing 118
MacHeaders option 128
Macintosh Programming Primer 7
Macintosh Programming Secrets 7
Macintosh Revealed 7
Macintosh Toolbox
    globals 125
    initialization calls 126
Macintosh Toolbox routines
    calling 119-125
    in assembly language 162
    passing arguments to 120
    special cases 121
Macsbug 127, 151, 152, 413
Macsbug Symbols 127, 413
MacTech Quarterly 8
MacTraps
    in code resources 87
    in drivers 80
MacTutor 7
main
    for code resources 86

for drivers 78
MakeClipboard
CApplication 231
MakeDecorator 288
CApplication 231
MakeDesktop 289
CApplication 230
MakeMacWindow
CWindow 394
Make... 119
making applications. 74-76
making code resources 84-92
making desk accessories 76-84
making device drivers 76-84
making drivers 76-84
Match Words option 100
MaxApplZone 126
MBDF resource 85
MC68000
assembly langauge 156
MC68000 8-, 16-, 32-Bit Microprocessors User's Manual, Sixth Edition 155
MC68020
assembly langauge 157
MC68020 32-Bit *Microprocessors User's Manual, Third Edition* 155
MC68020 option 128
MC68030 Enhanced 32-Bit Microprocessor User's Manual 155
MC68851 Paged Memory Management Unit User's Manual, Second Edition 155
MC68881
assembly langauge 157-158
MC68881 option 128
MC68881/882 Floating-Point Coprocessor User's Manual, Second Edition 155
MDEF resource 85
member 178, 187, 194
memberD 194
membership
in a class 187
MemoryReplenished
CApplication 235
MemoryShortage
CApplication 235
MenuKey 378
MenuSelect 291
Merge option 91
merging code resources 91
message 177
method
monomorphic 189
methods 178
calling 189
defining 188
inherited 189
using 188
MF Attrs 75
Monitor 151
monomorphic method 189
MoreSlots
CCluster 272
mouse tracking 219
Move
CWindow 397
MoveDown
CList 322
MoveOffScreen
CWindow 397
MoveToIndex
CList 322
MoveUp
CList 322
MPW 172, 173
Multi-File Search option 101
MultiFinder 76, 236
MultiFinder attributes 75-76
MultiFinder-Aware 75
multiple entry points 162
multiple entry points into functions 161
nAppleTalk 122
New 94, 186, 194
NewFile
CDocument 303
Notification Manager 7
Notify
CApplication 232
CBureaucrat 256
CDocument 302

NthItem
CList 322
numbers
sizes of 132
Numerical Recipes in C 6
OBJ_WINDOW_KIND 393
object 177
object code 68, 115
object reference 184
object-oriented programming 177-182
objects
as handles 191
circular references 185
creating 186
declaring 186
deleting 186
direct 189
in code resources 183
in desk accessories 183
instance variables 191
method calling 189
methods 188
this (implict variable) 188
using 183
oConv 173
Offset
CCluster 273
CControl 279
CPane 336
OFFSET 159
once-only headers 110
oops 183, 194
oops.h 194
oopsA4 183, 194
Open
CDataFile 284
CFile 318
Open Selection 95
OpenDocument
CApplication 237
OpenFile
CDocument 303
Open... 94
options
compiler 129
search 100
Options... 127, 129
origin 212
OSType 121
Out 145, 431
override 179
PACKED ARRAY[1..4] OF CHAR 121
Page Setup... 98
PageCount
CDocument 304
panes 207
drawing in 208
sizing 209
panorama 211
panoramas 211-213
ParseItemString
CBartender 249
partition size 75-76
Pascal
C routines that act like 123
callback routines 123
calling routines indirectly 124
strings 121
Pascal calling conventions 168
Pascal calling sequence 168
Pascal function entry 169
Pascal function exit 169-170
pascal keyword 437, 440
Pascal strings 438
Perform 236, 238, 239
CChore 264
PickFileName
CDocument 305
Place
CPane 336
PlaceNewWindow
CDecorator 288
Plauger 131
pointer arithmetic 440
points
passing as arguments 121, 133
polymorphism 180
portability 131
Unix 133
position 212
PostAlert

CError 314
pragmas 442
precompiled headers 117
creating your own 119
Precompile... 118
`Preload`
CApplication 236
`Prepare` 311
CControl 279
CDesktop 293
CPane 338
CView 391
CWindow 397
`PrepareToPrint`
CPane 338
`Prepend`
CList 322
`printf` 451
Printing manager routines, calling 122
`PrintPage`
CPane 338
CPanorama 346
CStaticText 373
`PrintPageOfDoc` 354
CDocument 304
`PrintPageRange`
CPrinter 354
Print... 98
`PrivateChanged`
CClipboard 269
`ProcessEvent` 235
Profile option 128, 413
Programmer's Guide to MultiFinder 75
Programmers Introduction to the Macintosh Family 7
project tree 109
projects
as libraries 171
component size limits 71
components of 69
resource files 71
running (applications) 76
THINK C 2.0 & up
compatibility 9
types of 68
prototypes See: function prototypes
`PtoCstr` 121
`PutData`
CClipboard 269
`PutGlobalScrap`
CClipboard 268
Quick Draw globals
in code resources 87
in drivers 80
`Quit`
CApplication 237
CDirector 297
rainy day fund 219, 226
`ReadAll`
CDataFile 285
`ReadSome`
CDataFile 285
`ReallyVisible`
CDesktop 290
CPane 335
CView 387
`Redo`
CTask 382
`Refresh`
CPane 337
`RefreshRect`
CPane 337
register usage
in assembly language 163
register variables 116
registration card 13
`RememberA0` 87
`Remove`
CCluster 272
CList 321
`RemoveButton`
CRadioGroup 360
`RemoveDirector`
CApplication 238
`RemoveIdleChore` 263
`RemoveMenu`
CBartender 246
`RemoveMenuCmd`
CBartender 248
`RemoveSubview`
CView 390

RemoveWind
CDesktop 292
Replace 100
Replace All 101
Replace and Find Again 100
replacing 99-103
with grep 105
Require Prototypes 129, 414
Resize
CWindow 397
ResizeFrame
CPane 338
CPanorama 346
CPicture 350
CStaticText 374
resource files 71
resources 67
Restore
CEnvironment 311
RestoreA4 87
RestoreA5 122
RestoreEnvironment
CPane 338
ResType 121
Resume 76
CApplication 236
CClipboard 267
CDirector 297
Revert 96
Ritchie 6, 131
root class 179, 184
rsrc 71
RTS 83, 161
Run 76, 377
CApplication 235
running applications 76
SANE utilities 123
Save 99
Save a Copy As... 99
Save All 99
Save As... 99
SBarActionProc
CScrollPane 368
SBarThumbFunc
CScrollPane 368
scanf 451
ScrapConverted
CClipboard 269
Scroll
CPanorama 346
CStaticText 372
scroll panes 213-214
ScrollTo
CPanorama 346
ScrollToSelection
CPanorama 346
CStaticText 372
Search 100
search options 100
searching 99-103
for patterns 103
for symbols 102
in the debugger 141
multiple files 101
non-printing characters 101
segments 72-74
deleting 74
in code resources 91-92
in drivers 84
moving 74
moving files among 73
unloading 84, 91
Select
CWindow 396
selected statement 140
selecting lines 97
SelectWind
CDesktop 292
SendBack
CList 322
Separate STRS option 75
SERD resource 122
serial driver 122
serial number 13
Set Breakpoint 142
Set Context 149
Set Project Type... 74, 77, 85, 183
Set Tabs & Font... 98
SetActClick
CWindow 395
SetActionProc
CControl 278
SetAlignment
CStaticText 374

SetBlockSize
CCluster 272
SetBounds
CPanorama 345
SetClickCmd
CButton 260
SetCmdText
CBartender 247
SetDimOption
CBartender 249
SetDropShadow
CBorder 254
SetEventMask 232
SetFontName
CStaticText 373
SetFontNumber
CStaticText 373
SetFontSize
CStaticText 373
SetFontStyle
CStaticText 373
SetFrameOrigin
CPane 335
SetInteriorSize
CBorder 254
SetLength
CDataFile 284
SetLineSpacing
CStaticText 374
SetMacPicture
CPicture 350
SetMark
CDataFile 284
SetMaxValue
CControl 278
SetMinValue
CControl 278
SetOverlaps
CScrollPane 367
SetPosition
CPanorama 345
SetPrintClip
CPane 335
SetRect, SetPt, etc. overhead
126
SetScaled
CPicture 350
SetScales
CPanorama 345
SetSizeRect
CWindow 395
SetSizeRect 397
SetStationID
CRadioGroup 360
SetStdState
CWindow 395
SetSteps
CScrollPane 367
SetTextHandle
CStaticText 373
SetTextMode
CStaticText 373
SetTextPtr
CStaticText 373
SetTextString
CStaticText 373
SetThickness
CBorder 254
SetThumbFunc
CScrollBar 362
SetTitle
CControl 278
CWindow 395
SetUnchecking
CBartender 249
SetUpA4 79
SetUpA4.h 87
SetUpA5 122
SetUpFileParameters
CApplication 231
SetUpMenus 243, 244, 246
CApplication 231
SetValue
CControl 278
SetWantsClicks
CView 387
SetWholeLines
CStaticText 374
SevereMacError 237
CError 313
SFPGetFile 238
SFPGetFile 231
SFSpecify
CFile 318

Shift Left 97
shift operators 440
Shift Right 97
Show
    CDesktop 290
    CPane 336
    CView 387
    CWindow 396
Show Condition 144
Show Context 149
ShowOrHide
    CWindow 396
ShowResume
    CWindow 396
ShowWind
    CDesktop 292
signatures (see file signatures)
signed 440
size
    of project components 69
size of an object 193
SIZE resource 75
sizeof 193, 440, 442
sizes of basic types 439
sizes of floating point numbers 132
sizes of int 132
sizes of numbers 132
sizing characteristics 209
Skip To Here 146, 432
sleep time 309
slots 271
smart linking 70
    turning off 71
Software Engineering in C 6
source files See: files
    compiling 115
    moving 111
Source window 137, 138, 139-141
Specify
    CFile 318
SpecifyHFS
    CFile 318
stack
    argument passing, C 166
    argument passing, Pascal 169
    function entry, C 166
    function entry, Pascal 169
    function exit, C 167
    function exit, Pascal 169
Stack frame too big 450
Standard C 6, 131
StartUpAction
    CApplication 236
statement markers 138
station 359
Status
    CClipboard 268
status panel 138
stdio library
    initializing 127
Steele 6
Step 44, 145, 431
Step In 45, 145, 431
Step Out 46, 145, 431
Stop 49, 145, 432
storage class specifiers 440
str2dec 123
stray pointers 76
string literals 438
strings
    converting to and from C and Pascal 121
    in Toolbox routines 121
    Pascal 121, 438
STRS component 69, 75
struct
    functions returning 167
struct declarations 440
structs
    displaying 54, 150
subclass 179
SubpaneLocation
    CView 391
superclass 179
supervisors 199
Suspend
    CApplication 236
    CClipboard 267
    CDirector 297
Suspend & Resume Events 76
SwitchFromDA
    CApplication 236
SwitchToDA

CApplication 236
syntax checking 116
system globals 125
SystemClick 291
tabs 98
TechAlliance 8
Technical Introduction to the Macintosh Family 7
TEIdle 257
temporary breakpoints 143
tentative definition 441
The C Programming Language 131
The C Programming Language, Second Edition 6, 437
THINK C Tree. 109
THINK Class Library
flow of control 200
THINK Customer Support 449
THINK_C (preprocessor symbol) 131, 442
this 188
ThrowOut
CFile 318
TMON 127, 151, 413
Toggle
CClipboard 268
Toolbox (See Macintosh Toolbox)
Toolbox routines 119
ToolScratch 89
Trace 46, 145, 432
TrackControl 259, 261, 280
TrackMouse
CView 388
trap intercept routines
in drivers 79
trees 112
trigraph sequences 442
type checking
setting options 129
type qualifiers 439
type specifiers 440
typedefs 185
Undefined symbols 449
Undo 96
CTask 382
undoing 219
union
functions returning 167
union declarations 440
Unix 133
UnloadA4Seg 84, 91
Update
CWindow 397
UpdateAllMenus
CBartender 249
UpdateDisplay
CClipboard 268
UpdateMenus 245
CBureaucrat 257
CEditText 308
UpdateUndo
CDocument 305
UpdateWindows
CDesktop 292
Use 2nd Screen option 137
Use Debugger 42, 136
UsePICT
CPicture 350
value column 50
variable-length argument lists 167-168
views 198
virtual 193
visual messages 196
void 130
volatile 439
WaitNextEvent 257, 309, 379
WantsActClick
CWindow 395
WDEF resource 85, 86
window coordinates 211
window decorator 231
window kind
of CWindow windows 393
windows 207
WindToFrame
CPane 339
WindToFrameR
CPane 339
Wrap Around option 100
WriteAll
CDataFile 285

WriteSome
CDataFile 285
x80tox96 123
x96tox80 123
XCMD resource 85
XFCN resource 85
Zone
increasing 126
Zoom
CWindow 397

License Agreement

License Agreement

This manual and the software described in it were developed and are copyrighted by Symantec Corp. (Symantec) and are licensed to you on a non-exclusive, non-transferable basis. Neither the manual nor the software may be copied in whole or in part except as follows:

1) You may make backup copies of the software for your use provided that they bear Symantec's copyright notice.

2) You have the right to include object code derived from the libraries in programs that you develop using the software and you also have the right to use, distribute and license such programs to third parties without payment of any further license fees, so long as a copyright notice sufficient to protect *your* copyright in the software in the United States or any other country is included in the graphic display of your software and on the labels affixed to the media on which you software is distributed.

You may not in any event distribute any of the source files provided or licensed as part of the software. You may use the software at any number of locations so long as there is no possibility of it being used at more than one location at a time.

Symantec's Plain Language License Statement

Symantec is concerned with how you copyright your software only in the case where you use object code of libraries which Symantec provides in source form (MacTraps does not fall into this category). These libraries may be included in your program so long as a copyright notice that will protect *your* copyright in the software is in the "About box" of your software and on the disk labels, as specified in the license agreement. You are not required to include a specific Symantec copyright notice except if your copyright does not satisfy the above requirement. This is only an explanation of the License Agreement. All terms and conditions of the License Agreement apply.

Limited Warranty on Media and Manuals

If you discover physical defects in the media on which this software is distributed or in the manuals distributed with the software, Symantec will replace the media or manuals at no cost to you provided that you return the defective materials along with a copy of your receipt to Symantec or to an authorized Symantec dealer during the 60-day period following your receipt of the software.

Limited Warranty on the Product

Symantec warrants that the software will perform substantially as described in the User's Manual. If within 60 days of receiving the software, you give written notification to Symantec

of a significant, reproducible error in the software which prevents operation, and provide a written description of the possible problem along with a machine readable example, if appropriate, Symantec will either provide you with corrective or workaround instructions, a corrected copy of the software, a correction to the User's Guide and Reference Manual, or Symantec will refund your purchase price upon return of all copies of the software and documentation together will a copy of your receipt. This warranty extends only to you and shall be void if the software has been tampered with, modified, or improperly used, or if the software is used on hardware other than the Apple Macintosh™ Computer.

EXCEPT FOR THE LIMITED WARRANTY DESCRIBED ABOVE, THERE ARE NO WARRANTIES TO YOU OR ANY OTHER PERSON OR ENTITY FOR THE PRODUCT EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OR MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE. ALL SUCH WARRANTIES ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. Some states do not allow the exclusion of implied warranties or limitations on how long they last, and you also may have other rights that vary from state to state. IN NO EVENT SHALL SYMANTEC BE RESPONSIBLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR SIMILAR DAMAGES OR LOST DATA OR PROFITS TO YOU OR ANY OTHER PERSON OR ENTITY REGARDLESS OF THE LEGAL THEORY, EVEN IF WE HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you. The warranty and remedies set forth are exclusive and in lieu of all others, oral or written, express or implied.

SYMANTEC MAKES NO WARRANTY OF THE PERFORMANCE OF THE LIBRARIES WHEN USED IN YOUR SOFTWARE. YOU AGREE TO INDEMNIFY SYMANTEC FROM ALL CLAIMS BY THIRD PARTIES ARISING IN CONNECTION WITH THE USE OF YOUR SOFTWARE.

General Terms This license states the entire agreement between the parties and supersedes all other communications between the parties relating to this License, which shall be governed and construed in accordance with the laws of the State of California. You agree to bring any proceeding to enforce or construe this License or involving the performance of the software only in a federal or state court residing in the State of California. The prevailing party in any such proceedings shall be entitled to recover its attorneys' fee and litigation expenses in addition to other appropriate relief. If any provision of this License by Symantec shall be held to be unenforceable such holding shall not affect the enforceability of any other provision hereof. Waiver of any breach of this License by Symantec shall not be considered a waiver of any other or subsequent breach. The licensed software is a unique and valuable asset of Symantec and Symantec has the right to seek whatever equitable and legal redress which may be available to it for your breach of the provisions of the License.

SYMANTEC™
Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014
(408) 253-9600

THINK C is a trademark of Symantec Corporation. © 1989
Symantec Corporation. All rights reserved. Printed in U.S.A.

05-30-00021
LSC002 7/89